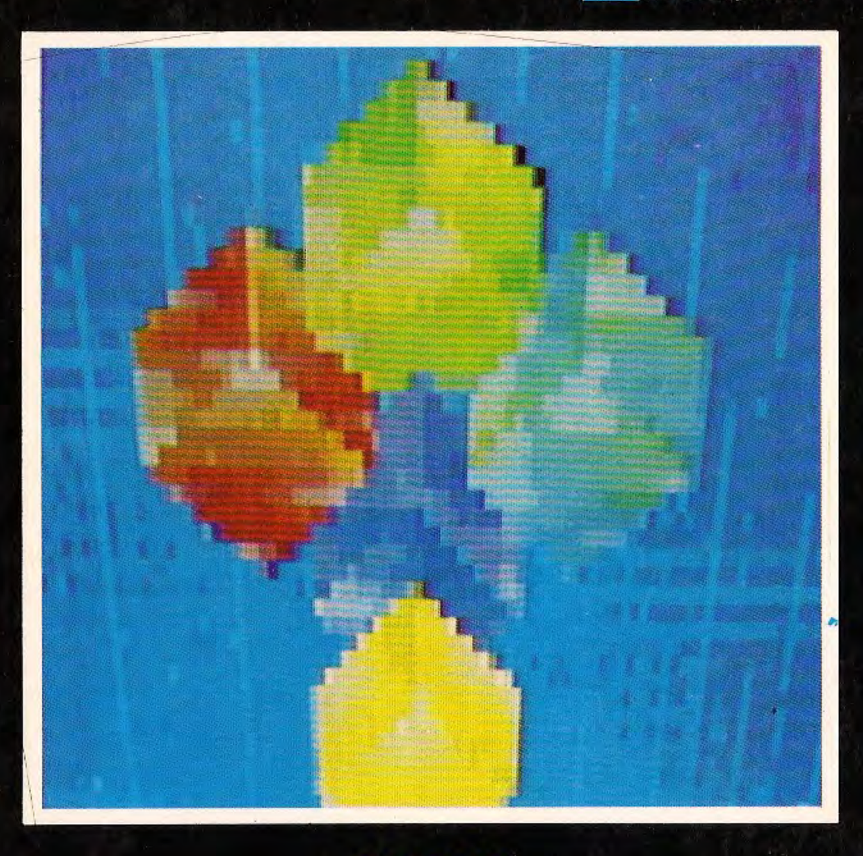
BIBLIOTECA BASICA INFORMATICA

INTRODUCCION AL PASCAL

8

programación estructurada



INGELEK

BIBLIOTECA BASICA INFORMATICA

INTRODUCCION AL PASCAL

8

programación estructurada

INGELEK

Director editor:
Antonio M. Ferrer Abelló.

Director de producción:
Vicente Robles.

Coordinador y supervisión técnica:
Enrique Monsalve.

Colaboradores:
Angel Segado.
Patricia Mordini.
Margarita Caffaratto.
Marina Caffaratto.
Francisco Ruiz.
Jorge Juan Monsalve.
Beatriz Tercero.
Fernando Ruiz.
Casimiro Zaragoza.

Diseño:
Bravo/Lofish.

Dibujos:
José Ochoa.

ISBN del tomo: 84-85831-39-X
ISBN de la obra: 84-85831-31-4
Fotocomposición: Pérez Díaz, S. A.
Imprime: Héroes, S. A.
Depósito Legal: M-37.689-1985

INDICE

PROLOGO

EN el mundo de los ordenadores personales los lenguajes de programación que se disputan el primer puesto son, prácticamente, sólo dos: el popularísimo BASIC y el más aristocrático Pascal.

A decir verdad, en el segmento de los microordenadores profesionales, para entendernos, aquellos que están dotados del sistema operativo CP/M (en las máquinas de 8 bits) o MS DOS (máquinas de 16 bits) encontramos también otros lenguajes, bien conocidos en la informática tradicional, como el FORTRAN o el COBOL, dedicados el primero a la programación científica y el segundo a la de gestión. Incluso se ha llegado a introducir en el PC IBM el ya venerable RPG II, lenguaje sin duda interesante de tipo "no de procedimiento", es decir, que el programador define las especificaciones del problemas más que las instrucciones específicas; se puede considerar que en él la lógica de control queda sobrentendida por lo menos a grandes rasgos. Está orientado, por su propia naturaleza, al tratamiento de datos por "lotes" (en inglés "batch"), es decir, en grandes masas consecutivas. ¿Puede haber algo más extraño al auténtico espíritu —interactivo, transaccional y en tiempo real— del ordenador personal? Para quien no conozca estos términos, recordamos que se refieren, respectivamente, al diálogo hombre-sistema, al tratamiento esporádico de datos únicos y a la inmediatez de los resultados (esto, naturalmente, si es que la lentitud de ciertos intérpretes BASIC lo permite).

Con las herramientas de programación mencionadas (COBOL, FORTRAN y RPG) se ha realizado el reciclaje del enorme volu-

men de software creado en los años sesenta y setenta con una cierta facilidad. Este software se creó para ordenadores grandes y miniordenadores, especialmente en el campo comercial para contabilidad, facturación, almacenaje y demás grises pero útiles aplicaciones. Y es una pena porque este proceso quizá haya impedido a bastantes programadores ponerse al día y concentrar sus esfuerzos en las nuevas aplicaciones para ordenadores personales: gestión de bases de datos descentralizadas, hojas electrónicas ("spreads heet") y, ¿por qué no?, inteligentes juegos computerizados.

Pero no divaguemos más y volvamos a los ordenadores personales "de verdad". Dos son, fundamentalmente, los motivos que hacen del BASIC y del Pascal los lenguajes más adecuados a este nuevo espíritu.

- la interactividad,
- la sencillez y la rapidez de uso.

Para ser exactos, la primera virtud es más propia del plebeyo BASIC que del noble Pascal, y está ligada principalmente al carácter interpretativo del primero, aunque hay que recordar que también existen BASIC compilados. Gracias a la interpretación, es decir, a la traducción directa durante la ejecución, característica común también al LOGO y demás esotéricos lenguajes, como el FORTH y el APL, es posible probar rápidamente un programa sin problemas ni complicaciones. En cambio, el Pascal es compilado, lo que comporta el uso de multitud de herramientas: un editor para redactar los programas, un enlazador-cargador ("linker-loader") para conjuntarlos y cargarlos en el área de trabajo y un compilador para traducirlos al lenguaje máquina.

En honor a la verdad, en ciertas implementaciones del Pascal que parten del excelente UCSD Pascal, realizado por el profesor Bowles, de la Universidad de San Diego, en California (UCSD significa precisamente University of California at San Diego) se ha hecho un gran trabajo de simplificación para hacer más fáciles y claras estas aburridas operaciones. Sin embargo, queda el hecho de que la fase de edición, independiente de la de compilación, sigue siendo un mal necesario, con el agravante, además, de tener que reeditar el programa desde el principio cada vez que el compilador dé un error de sintaxis (también en BASIC "syntax error" es un mensaje que sale cada que vez que se viola alguna regla ortográfica, pero no es necesaria ninguna maniobra para cambiar de "ambiente", dado que siempre se permanece en un mismo y único entorno).

Además, algunos críticos autorizados han llegado a sostener —aunque no compartimos plenamente su extremismo— que "... la reintroducción de la compilación en el ámbito de los ordenadores personales realizada por el Pascal... ha sido un error".

¿Por qué hablar entonces de programación estructurada, basándose en el Pascal, en un libro de divulgación principalmente dirigido al usuario final? Pues porque el Pascal posee por lo menos una virtud definitiva que el BASIC ni siquiera puede soñar:

La estructuración

Es decir, un planteamiento formal claro y riguroso inspirado en los más modernos y sanos principios del arte de la programación.

Los consabidos críticos autorizados tildan al BASIC de lenguaje "barullo", con las pésimas costumbres del vetusto FORTRAN (el COBOL, en cambio, ya les resulta más presentable, pues tiene una cierta pinta de lenguaje semiestructurado...), tan es así que hoy día en la mayoría de nuestras universidades no se enseña más que Pascal y un poco de FORTRAN, mientras que el extendidísimo BASIC queda olvidado, con el consiguiente peligro, también es verdad, de formar informáticos aspirantes al desempleo, dado que no saben programar ni en BASIC ni en COBOL. Pero esto es otro tema, y, si en todo caso, la programación estructurada no tiene ninguna culpa, ni con ello se menoscaba su excelente capacidad pedagógica y formativa.

Efectivamente, el Pascal fue ideado originariamente con finalidades didácticas. El padre de este lenguaje, el suizo Niklaus Wirth, profesor del Instituto Politécnico de Zurich (que además ha presentado recientemente una nueva versión, expresamente modular, denominada Modula 2), siempre ha declarado que se considera esencialmente un docente. En consecuencia, el Pascal, en alguna de sus implementaciones, no está ni siquiera dotado de opciones adecuadas para la gestión de archivos sobre memorias de acceso aleatorio (disco) mientras que, sin embargo, el "basto" BASIC posee por lo menos aquellas funciones necesarias para la vida cotidiana.

Hacer mención de esta y otras carencias, como, por ejemplo, la ausencia, aunque solamente en el Pascal estándar de Wirth, de variables tipo cadena, casi imprescindibles en la manipulación de datos alfanuméricos, no significa querer dar pie a la polémica, y aún menos desanimarles de la lectura de los capítulos siguientes. Nuestro interés es que muchos usarios convencidos del BASIC, que perseveran en el vicio de añadir desaforadamente líneas a los programas, preocupándose únicamente de que éstos corran (lo que acaba ocurriendo después de pasar infinitas penalidades, siempre proporcionales a la prisa con la que se escribe la prime-

ra versión), aprovechan una buena ocasión de aprender por lo menos reglas de buen estilo. Y no sólo esto.

En efecto, los cánones de la programación estructurada pueden resumirse en los tres siguientes:

- definir una serie de construcciones estándar sobre las que articular, por bloques, la totalidad del programa;
- subdividir un problema complejo en subproblemas más sencillos (principio de "divide y vencerás"). Se traduce en el plantamiento top-down (de arriba a abajo) de los programas;
- determinar una tipología y unas estructuras de datos adecuadas para el problema considerado.

Aprender, y mejor si es con espíritu crítico, estos tres principios le sirve a cualquiera, porque se acostumbra a trabajar de una forma ordenada y rigurosa que facilita enormemente la comprensión de lo que se hace. Y esto es un aspecto importantísimo en especial para aquellos que trabajan profesionalmente en el mantenimiento de software.

Puede que oiga decir que todo lo expuesto no tiene validez para el BASIC, puesto que este "tosco" (aunque fácil y eficaz) lenguaje no posee las estructuras IF/THEN/ELSE, DO/WHILE, REPEAT/UNTIL, etc., características; no haga caso. Este tipo de comentarios están completamente fuera de lugar, puesto que reducen la programación estructurada solamente a su primer aspecto, aquel que, en suma, es el más externo, formalista y, llevado al límite, bizantino. El propio Wirth considera que la precisión formal es un medio, no un fin. Resulta extremadamente significativo al respecto el feliz título de un excelente texto de nuestro "gurú" suizo: "Algoritmos + estructuras de datos = programas".

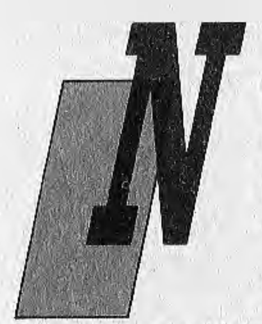
La subdivisión en módulos y la estructuración de los datos se revelan como el aspecto más potente y característico del estructuralismo en la programación. Y, como intentaremos demostrar en un próximo volumen de la BBI (programando como es debido), estos dos principios pueden, al menos en parte, ser emulados con los sencillos medios que el BASIC pone a nuestra disposición.

Será más fácil para quienes se enamoren del Pascal (los usuarios de este elegante lenguaje son muchísimos). Quienes permanezcan fieles al BASIC tendrán mucho que aprender, no pudiendo ignorar que existen ya, y se espera que aumente su número, dialectos BASIC planteados cada vez de forma más parecida al Pascal.

CAPITULO I

TOP-DOWN: PROGRAMANDO DE ARRIBA A ABAJO

Los orígenes



No debemos engañarnos: la programación estructurada desilusiona a veces, aunque quizá sólo en una mínima parte, a quien se "casa" con ella. Por eso no queremos tampoco mitificarla demasiado (a pesar de lo cual le tenemos que confesar que nos encanta).

Para hacernos una primera idea empezaremos por una de sus más célebres peculiaridades: la de la programación descendente (llamada en inglés "top-down"), que consiste en empezar analizando el problema en sus líneas generales, para después, poco a poco, dilucidar los casos particulares. Todos los pasos, naturalmente, deben enlazarse de una forma perfecta, con una serie de hábiles y precisos encajes, como si fuera un puzzle.

La ventaja de este planteamiento reside, en primer lugar, en el hecho de que se ofrece una visión de conjunto, panorámica, más fácil de dominar. Como decía uno de los máximos representantes del estructuralismo, "el hombre tiene la cabeza pequeña", y por tanto, se cansa si tiene que dominar asuntos demasiado amplios o complicados. Si lo pensamos bien, esto no es una novedad. Los dibujantes trazan desde hace siglos en hojas separadas las vistas de conjunto y los dibujos de detalles. Los "estructuralistas" han tenido que hacer su guerra particular por ser la Informática, en el fondo, una disciplina joven. Durante mucho tiempo (los malintencionados añadirán: incluso ahora, en el oscuro rincón de los viejos centros de elaboración de datos...) se han desarrollado

programas kilométricos escritos como un todo, y que tenían al final un desenvuelto

```
30000 IF RESP$="Y" THEN GOTO 120
```

Peor todavía: se han escrito programas en los cuales las condiciones de final de ejecución estaban en un imprevisible punto intermedio. En estas condiciones, seguir el discurrir de un largo programa, especialmente si no ha sido escrito por nosotros, se vuelve algo realmente trágico, y comporta un continuo enrollar y desenrollar el listado.

Estructuras de control y estructuras de datos

Otra crítica de los estructuralistas al modo tradicional de escribir software se refiere al estilo de programación "espagnetti" (programas desordenados). Veamos un sencillo ejemplo en BASIC:

```
300 IF A>B THEN 330
310 C=B-A
320 PRINT A,B,C:GOTO 340
330 C=A-B:GOTO 320
340 <Resto del programa>
```

Para ser sinceros, creemos que nos hemos "pasado" un poco con el ejemplo. De todas formas, en lenguajes como el Pascal o incluso en algunos dialectos del BASIC dotados de la construcción. IF-THEN-ELSE, la cuestión se reduce a:

```
300 IF A>B THEN C=A-B ELSE C=B-A
310 PRINT A,B,C
320 <Resto del programa>
```

Que para quien no lo sepa, se lee de una forma así de seguida y elocuente: si (IF) A es mayor que B, entonces (THEN) calcula C como diferencia entre A y B; si no (ELSE) calcúlalo como diferencia entre B y A. Sencillo, ¿verdad?

Pero sigamos con el tratamiento descendente. La programación estructurada quiere constituir un conjunto orgánico de reglas de cara a la realización de "buenos" programas. Este adjetivo, indudablemente genérico, se verá mejor definido, especialmente en los capítulos siguientes, cuando demos ejemplos concretos de este moderno planteamiento. Bajo una visión general, la programación estructurada presenta tres aspectos:

- sintaxis,
- semántica,
- pragmática.

El primero está en relación con los lenguajes concretos de programación que se inspiran explícitamente en principios parecidos: después del "progenitor" (Algol) se habla sobre todo de Pascal, Ada, Modula 2 y "C"; hay algunos atisbos también en COBOL y en algunos BASIC.

En cuanto a la semántica, o sea, al significado de las palabras o al encuadramiento conceptual de las "estructuras", tenemos que subrayar desde ahora que, para estructurar no se deben de tomar sólo en cuenta, como sucede común y superficialmente, las estructuras de control, es decir, de gobierno del flujo de las instrucciones, sino también las ESTRUCTURAS DE DATOS íntimamente ligadas a las primeras.

La pragmática, por último, se refiere a las consecuencias prácticas, con prisa por materializar los "principios sagrados" en la programación de todos los días. Este aspecto enlaza, obviamente, con la exposición de la sintaxis del Pascal. En términos generales, la pragmática tiene que ver con la estructuración de la "arquitectura" de un programa, centrada fundamentalmente en el desarrollo top-down (descendente) de los programas y de las estructuras dadas.

Una gran polémica: GOTO sí, GOTO no

El fundamento más célebre y controvertido de la programación estructurada es el anti-GOTO, o más finamente, "GOTO-less programming", junto al "ego-less programming", es decir, la programación clara para todos y bien documentada. El "GOTO-less programming", entendido radicalmente, significa suprimir para siempre la instrucción GOTO, de salto incondicional. Alguien dirá: ¿no son los saltos la sal misma de la programación? Además, ¿cómo se puede evitar si está prácticamente integrada con las operaciones de decisión (IF) y, por lo tanto, con los saltos condicionales?

Para que nos entendamos mejor, hay que señalar que estamos hablando de lenguajes de programación de alto nivel, o sea, de aquellos "orientados al problema". En el lenguaje máquina los saltos incondicionales son inevitables (aunque hay quien está proyectando CPUs de arquitectura inspirada en el estructuralismo), pero pasando a un nivel más elevado, más cercano a nuestra manera de razonar y de hablar, deberíamos poder abolirlos. Respec-

to a la manera de hacerlo hemos visto un pequeño ejemplo en el programa anterior, usando la construcción ELSE.

Entre los primeros que arremetieron contra el vituperado GOTO está Esdger W. Dijkstra, que en 1968, en una famosa carta enviada a la revista de la ACM (Asociación de Cálculo Automático USA), proclamó la necesidad de eliminarlo en todos los lenguajes de alto nivel que pretendieran ser dignos de este nombre.

El título de la comunicación era como un latigazo: "GOTO considered harmfull", algo así como "Ese maldito y pernicioso GOTO" (en traducción libre, claro). En ella vertía duros juicios contra el vetusto FORTRAN (lenguaje para usos científicos) y el PL/1 (lenguaje de uso general, más moderno, pero que tuvo poco éxito): al primero, lo definía como "enfermedad infantil", y al segundo le auguraba "una muerte fatal".

En correspondencia con el "GOTO-less", se han desarrollado varios lenguajes en los cuales la instrucción GOTO no existe (por ejemplo, el BLIS, o, por citar otros conocidos también en los ordenadores personales, el LOGO y el FORTH). Hay algunos, sin embargo, que aunque sean estructurados lo toleran: por ejemplo, el lenguaje C acepta frases con saltos incondicionales, y el mismo Pascal incluye la instrucción GOTO, a pesar de que desaconseja decididamente su uso. Efectivamente, además de imponer que la etiqueta de destino del salto se declare explícitamente (se tiene el equivalente de un GOTO 150 del BASIC, pero con el inconveniente de tener que definir previamente 150 como etiqueta "label") es habitual que en los manuales de Pascal la instrucción GOTO sea tratada en último lugar, como diciendo: "si verdaderamente no sois capaces de prescindir de ella, aquí la tenéis, para casos extremos".

Esta "prohibición" del GOTO le parece todavía a mucha gente una manía. De todas formas, como comentamos antes, no agota los cánones de la programación estructurada. Estos, lo repetimos de nuevo, son tres:

- desarrollo descendente (top-down),
- modularidad, o sea, subdivisión en pequeñas partes,
- estructuración de los datos, además de los programas.

Dijkstra, en "Structured programming" (texto de obligada referencia, escrito junto con otros dos "magos" de la informática: Dahl y Hoare), hace hincapié en que para dominar un tema tan complejo la única esperanza consiste en la estrategia del divide et impera ("divide and conquer", que dicen los ingleses) Todo se resume en el uso disciplinado de ciertas estructuras de control que vamos a reseñar inmediatamente, no sin antes repetir que, después de todo, deben integrarse íntimamente con estructuras paralelas de datos.

Un teorema elegante

Tres estructuras (o construcciones) elementales son los auténticos modelos de pensamiento, base de una programación ordenada:

- secuencia;
- selección, a la que corresponde la construcción IF-THEN-ELSE;
- iteración, asociada a la construcción DO-WHILE.

Ninguna de ellas requiere una instrucción GOTO o similar. El soporte teórico de este conjunto es el famoso teorema de Jacopini-Böhm, que demuestra que cada programa, por complejo que sea, puede desglosarse en las tres estructuras mencionadas (Fig. 1).

Para explicarlas daremos por descontado que se conocen los diagramas de flujo (flow-chart), es decir, la representación gráfica —con rectángulos, rombos y otros símbolos análogos— de un programa. De todas formas, para beneficio de ignorantes y dejados que no quieran repasar los anteriores volúmenes de la colección, recordaremos que: un bloque rectangular contiene una o más instrucciones, o bien expresa un subprograma, procedimiento o

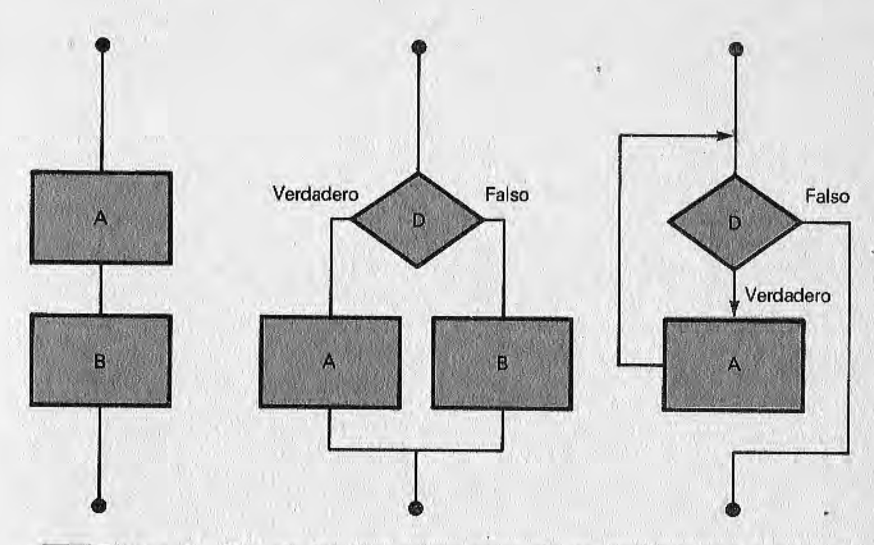


Figura 1.—Uso de las tres estructuras de control básicas según el Teorema de Jacopini-Böhm (secuencia, selección, iteración).

subrutina (en este caso el rectángulo suele tener dos rayas verticales en los laterales), cuyo nombre será situado en su interior; un rombo representa un bloque de decisión; contiene en su interior una pregunta o el test de una determinada condición (comparación o similar) y tiene dos salidas, correspondientes la primera a "verdadero" (o "SI") y, la segunda, a "falso" (o "NO").

Un bloque cualquiera puede, a su vez, resumir (es la idea del top-down) un conjunto de bloques más o menos intrincados en los que está subdividido. En la figura 1 hemos denominado A y B a las sentencias de los rectángulos, y D a la decisión genérica o pregunta de los rombos.

Fiémonos de Jacopini & C. y examinemos rápidamente las tres estructuras propuestas por ellos, una tras otra. La primera es banal; corresponde sencillamente al hecho de que los programas derivan de la secuencia de instrucciones. La estructura de selección IF-THEN-ELSE ya la hemos examinado en la práctica; unida al desarrollo top-down, permite una visión panorámica notable. Por ejemplo, en un BASIC estructurado se escribirá:

```
300 IF <cond> THEN GOSUB 1000 ELSE GOSUB 1500
```

donde <cond> representa una condición genérica a verificar.

Supongamos que la subrutina 1000 calcula los descuentos para los grandes clientes y la 1500 para los ordinarios; queda entonces claro lo que el programa, a grandes líneas, realiza. Anticipemos que el Pascal, lenguaje de más alto nivel que el BASIC, aliña todo con una semántica más eficaz de los nombres de procedimiento. Tendremos así algo parecido:

```
IF <COND> THEN DESCBRUT ELSE DESCORD
```

donde "descbrut" y "descord" son, ciertamente, más elocuentes que los ininteligibles GOSUB del BASIC.

Finalmente, la tercera estructura DO-WHILE. Hay que decir que la iteración no queda ignorada del todo por el BASIC, pues dispone del bucle FOR-NEXT (heredado del DO del FORTRAN). Aunque cómodo, no es tan general como el DO-WHILE, que permite expresar prácticamente cualquier tipo de iteración. En el DO-WHILE debemos expresar una condición D: si D no se cumple salimos del ciclo; en caso contrario continuamos en él. En resumen: el ciclo se repite (DO) "mientras" (WHILE) se verifica la condición.

En cualquier planteamiento "GOTO-less" el programa entero está formado por bloques consecutivos que presentan un solo punto de entrada y un solo punto de salida, lo cual permite juntarlos fácilmente, como si se tratara de los vagones de un tren.

Las mayores ventajas de la subdivisión en bloques se obtienen cuando es posible su reutilización en otros programas: se convierten en las llamadas funciones o procedimientos de librería, que permiten un gran ahorro de tiempo.

La tendencia es, pues, crear un programa principal ("main program") dentro del cual los distintos procedimientos (como se les llama en Pascal, o subrutinas como se les llama en BASIC) son "llamados" sucesivamente. Al estar los procedimientos escritos aparte, cuando depuramos el programa es posible que sólo tengamos que modificar algún procedimiento defectuoso sin tener que tocar los demás ni el principal, o bien, más raramente, se puede revisar el flujo del principal sin tocar los procedimientos, especialmente si éstos se refieren a la elaboración de subrutinas (Fig. 2). Este es uno de los fundamentos de la llamada ingeniería del software (software engineering), que se ocupa de la construcción correcta y eficiente de los programas, además de buscar su mantenimiento de la forma más eficaz posible. Si lo pensamos detenidamente, este principio también lo llevan a la práctica los programadores de FORTRAN y BASIC gracias a una cautelosa dosificación de subrutinas. ¿Entonces...? Lo que el Pascal (y otros lenguajes) ofrece de más son unas herramientas de trabajo (tools) más definidas y potentes.

Hasta aquí la propaganda. Pero, ¿es realmente todo tan bonito como parece?

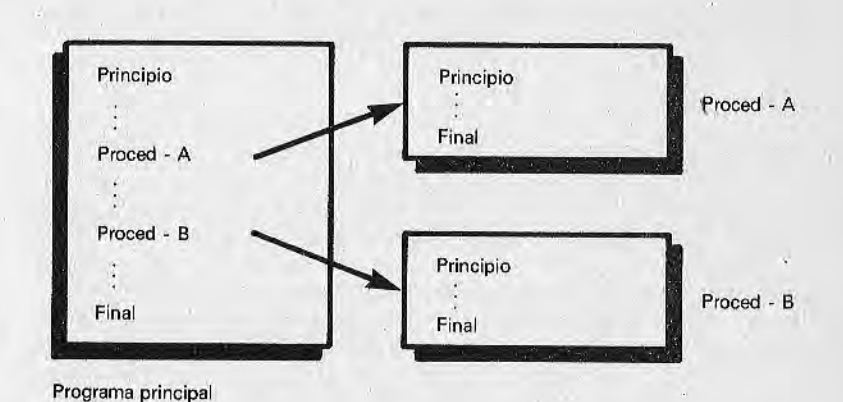


Figura 2.—Otro punto esencial de la programación estructurada es la subdivisión en módulos, bajo la forma de "procedimientos", reclamados desde el programa principal. El método ciertamente no es exclusivo de los lenguajes estructurados, pero éstos suministran "herramientas" más potentes.

Para contestar a esta peliaguda pregunta hay que hacer notar ante todo que, después de los furores iniciales, el extremismo estructuralista ha cedido un poco, ya sea por la práctica de los usuarios o por las teorías de otros estudiosos menos extremistas. Entre estos últimos debemos citar las autorizadas opiniones de David Knuth (ha escrito una especie de enciclopedia de algoritmos, aún inacabada, llamada "The art of programming"). Knuth advirtió el peligro de llegar a escribir programas oscuros y enrevesados, aun respetando de lleno las reglas básicas de la programación estructurada. Responsable de estos resultados no demasiado optimistas ha sido sobre todo una interpretación extremista del Teorema de Jacopini-Bohm. Efectivamente, si además de suprimir el GOTO nos limitamos dogmáticamente a las tres estructuras de la figura 1, pueden surgir fácilmente inconvenientes como los evidenciados en la figura 3, que, en la práctica, obligan a introducir códigos duplicados y/o desviadores (switchs en inglés). En ambos ejemplos hemos entendido la equivalencia como que entre los puntos "x" e "y" las situaciones son idénticas en el esquema original y en el estructurado correspondiente.

En el caso a), para obtener un diagrama de flujo equivalente, compuesto por las tres únicas estructuras básicas, se manifiesta en seguida la necesidad de duplicar el bloque A; de tal modo nos vemos reconducidos a la secuencia de A seguida de una estructura DO-WHILE, con los bloques B y A.

El segundo ejemplo, más complicado, consiste en un bucle con posible salida anticipada al verificarse dos condiciones ("p" y "q"). La reducción a módulos iterativos y de selección (IF-THEN-ELSE) se resuelve esta vez recurriendo al desviador (o "conmutador") SW; éste es una variable booleana, que sólo puede valer, por tanto, "true" o "false" (otras nomenclaturas equivalentes son: "1", "0"; "ON", "OF"; "activado", "desconectado"; "encendido", "apagado"). El desviador no se utiliza para ningún cálculo; sirve únicamente para desviar el flujo del programa cuando ocurren determinadas condiciones. En los diagramas de flujo la situación on/off de los desviadores generalmente está encerrada dentro de exágonos irregulares. Como se ve en la figura 3, SW está inicialmente desconectado, para activarse después al producirse una de las condiciones ("p", "q"). De esta manera, al inicio del nuevo ciclo, en el punto "z", SW conserva rastros de lo ocurrido en "p" o en "q", interrumpiéndose la iteracción (punto "y"), en caso de que alguna se hubiera cumplido. Para comprender la función de un desviador se puede imaginar un tren que, al llegar a un cambio de agujas, puede girar a la derecha o izquierda según un suceso precedente (por ejemplo, que hubiera pasado otro tren).

Volviendo a la figura 3 hay que reconocer que se ha conseguido descomponer todo en secuencias encerradas en una única

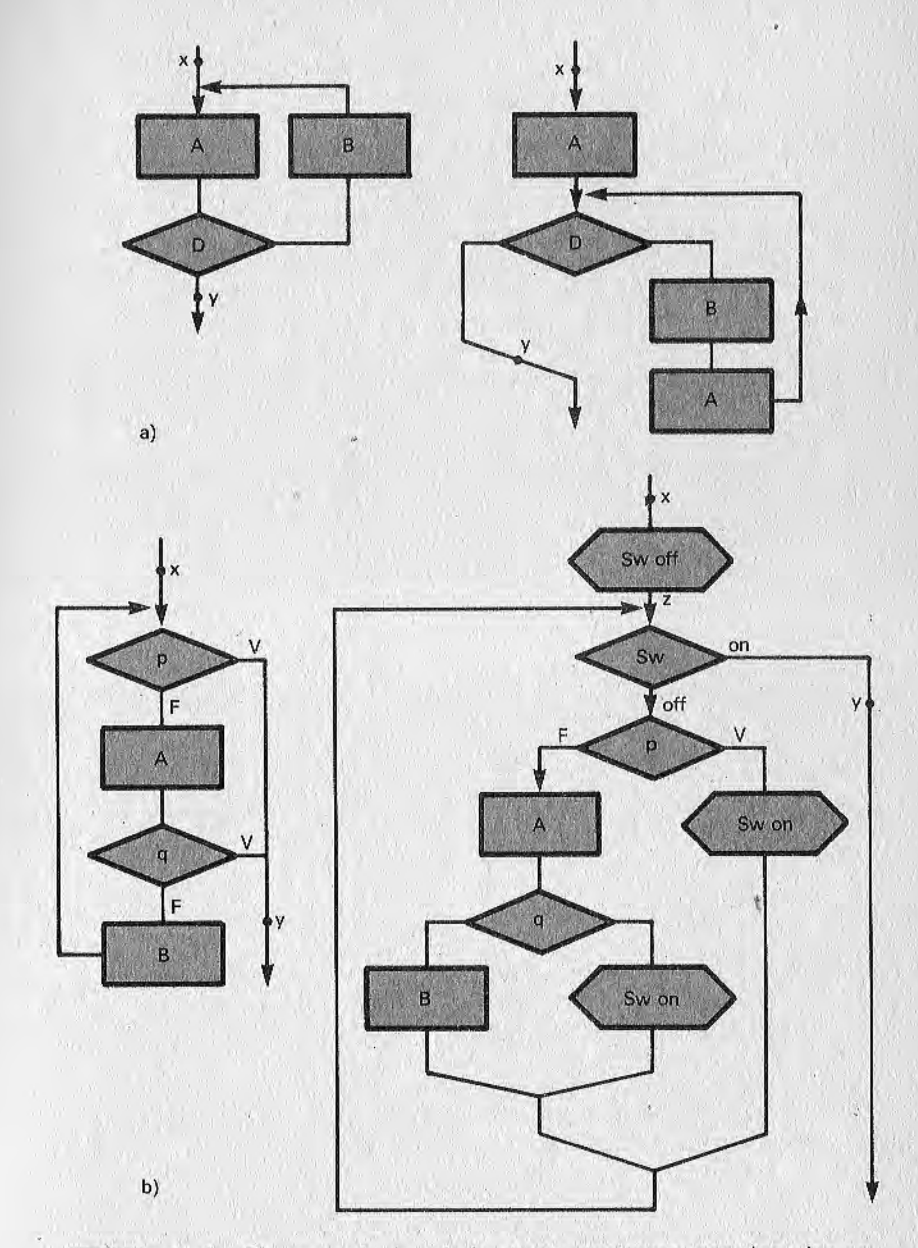


Figura 3.—El uso exclusivo de las tres estructuras mencionadas en el teorema de Jacopini-Böhm puede comportar redundancias: en a) ha sido necesario duplicar un bloque, en b) hubo que recurrir a desviadores (switchs).

estructura DO-WHILE (empieza con el rombo SW arriba), pero a un precio que supone casi contradecir uno de los presupuestos mismos del estructuralismo: la claridad (y podríamos proponer ejemplos en los que la situación sería trágica). La culpa de estas situaciones no es, evidentemente, de Jacopini, ni mucho menos del agradable Knuth; ellos se han limitado a demostrar y comprobar los efectos de un simple y honrado Teorema.

Ahora bien, ¿es cierto que nos debemos limitar a estas tres únicas estructuras? Ciertamente, no. La programación estructurada nos permite disponer de estructuras de control del flujo que, al menos sobre el papel, forman un repertorio para casi todas las situaciones. La figura 4 nos muestra las más típicas y difundidas. La sintaxis utilizada es genérica y no tiene una correspondencia exacta con ningún lenguaje en concreto.

En el DO-WHILE (haz mientras...) se abandona el bucle cuando la condición se deja de cumplir (es falsa), en tanto en el DO-UNTIL (haz hasta que...) ocurre al revés, saliendo de la iteración al cumplirse la condición.

Las construcciones REPEAT WHILE y REPEAT UNTIL funcionan como las dos anteriores, pero la verificación de la condición se hace al final del bucle, lo cual implica que, como mínimo, éste será recorrido una vez.

El paso de una construcción WHILE a una UNTIL, o viceversa, es inmediato, pues basta invertir la condición (A<B en lugar de A>B, por ejemplo) o usar el NOT. El Pascal normalmente se limita a las estructuras DO-WHILE y REPEAT UNTIL.

La construcción 5) se refiere a las salidas intermedias, en tanto la 6) es la clásica DO del FORTRAN y la conocidísima FOR del BASIC, por lo que no merece más comentarios; sólo observar que el equivalente en el Pascal estándar la variación del índice sólo puede ser en una unidad y no se permite modificar su valor en el bucle.

Finalmente, dos palabras sobre la penúltima estructura, ausente en la versión estándar del Pascal, pero introducida en algunas versiones comerciales y aceptada también por el mismo Wirth en su nuevo lenguaje (Modula 2). La cláusula EXITIF (en el lenguaje C se llama "break", denotando una ruptura anticipada del ciclo al cumplirse alguna condición intermedia) corresponde a una situación que, en BASIC, sería expresada como sigue:

```
3150 IF <CONDICION> THEN RETURN
.........
3200 RETURN:REM Final subrutina
```

Una estructura como ésta (tolerada por los puristas a pesar de que el EXITIF equivale a un GOTO gracias a que salta al final

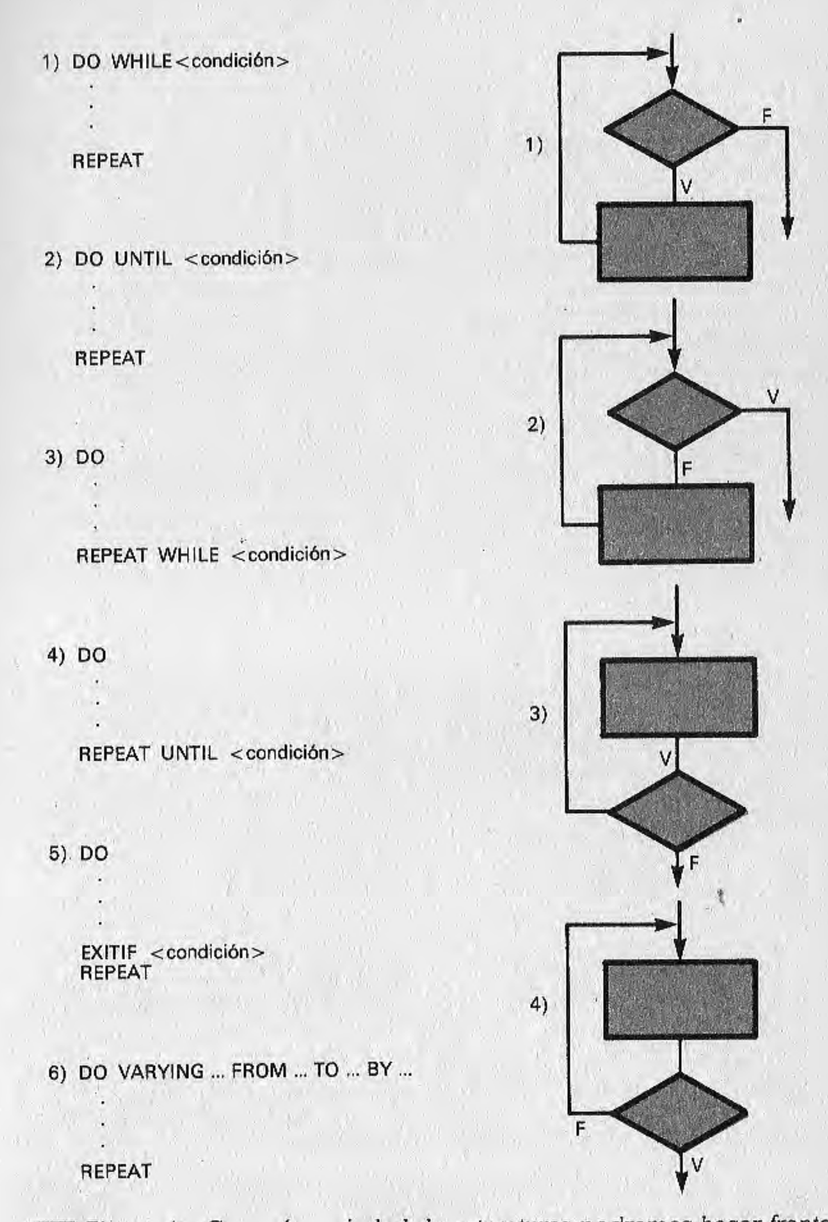


Figura 4.—Con más variedad de estructuras podremos hacer frente a los problemas surgidos con la limitación a las tres esenciales. En las primeras parejas la diferencia está en la condición de salida (V o F). El Pascal sólo adopta las construcciones DO-WHILE y REPEAT UNTIL. Es útil la posibilidad de salida anticipada (EXITIF, llamada a veces "break").

del procedimiento, salvando así el principio de que sólo exista un único punto de salida) resuelve problemas como los del ejemplo b) de la figura 3, permitiendo la interrupción de un ciclo al hacerse verdadero algún test interno a él; de esta manera se subdivide, de hecho, en distintas ramas.

Para completar el cuadro, debemos mencionar la estructura CASE. Actúa aproximadamente como la ON <condición> GOSUB xx, yy, zz del BASIC. CASE realiza un test con soluciones múltiples (superior a dos), para cada una de las cuales tiene definidas las acciones que hay que cumplir y/o los procedimientos a los que debe recurrir. En esencia es una ampliación de la IF-THEN-ELSE, con distintos anidamientos a niveles jerárquicos inferiores.

```
IF (condic. - 1) THEN
                 BEGIN
                  instrucción - 1
                  DO VARYING I FROM 1 TO 60
                     IF (condic. - 2) THEN instrucción - 2

                                       ELSE instrucción - 3
                     ENDIF
                  REPEAT
                 END
ENDIF
```

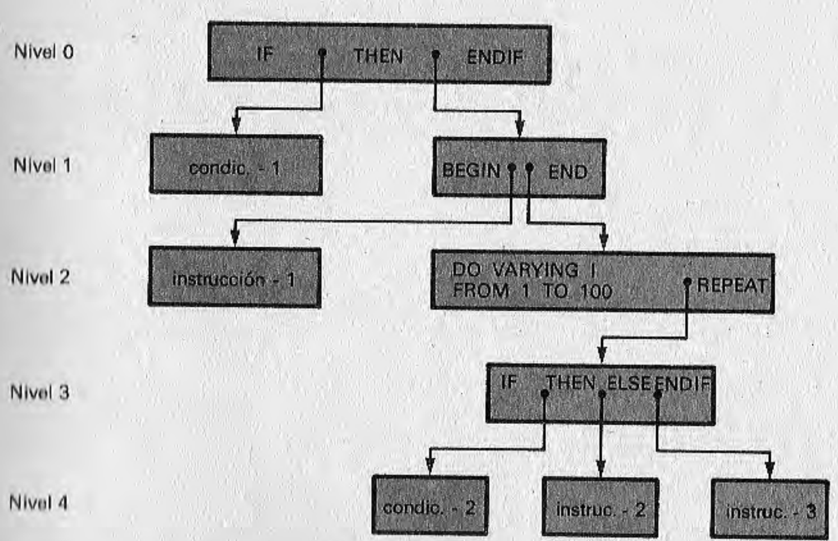


Figura 5.—El anidamiento de estructuras con distintos niveles jerárquicos es típica de los lenguajes estructurados. Se suele representar mediante "indentación" usando márgenes diversos.

En términos generales se expresaría como: IF <condición> THEN <expresión> ELSE IF... Esto enlaza con la idea del desarrollo topdown, que, además del aspecto mostrado en la figura 2, podría presentarse como se observa en la figura 5. Este simple ejemplo debería clarificar la idea del anidamiento de pequeños bloques unos dentro de otros (algo así como las muñecas rusas); en él se han representado los diferentes niveles de anidamiento gráficamente: en el más bajo encontramos el esqueleto de la construcción IF/ENDIF; en el nivel 1 se halla la condición-1 (que en ciertos casos podría implicar operaciones de más bajo nivel) adosada a la "caja" BEGIN... END, que, a su vez, encierra la instrucción-1 y una construcción DO-VARYING, que, a su vez...

En el transcurso de este libro nos acostumbraremos a este mecanismo y a la particular "indentación" (inglés: identation) con la cual se trata de evidenciar, mediante diferentes márgenes del texto, los distintos niveles (Fig. 5).

¿Será tedioso o divertido? Depende de los gustos. Util, desde luego.

CAPITULO II

MAS SOBRE EL TOP-DOWN: ESTRUCTURAS DE DATOS

Ser abstractos para resultar concretos

A menudo nos olvidamos que la elaboración de datos no se limita sólo al cálculo numérico, aunque los ordenadores hayan nacido con este fin (tanto es así que mucha gente sigue llamándolos calculadores). El cálculo matemático se inicia seriamente con el filósofo y matemático Blaise Pascal (no es una coincidencia: Wirth llamó Pascal a su lenguaje en honor de Blaise); inventó la "Pascaline", una máquina de sumar mecánica a base de ruedas, precisamente porque se apiadó del trabajo contable tan inhumano que su padre, recaudador real, realizaba. La historia de los computadores se refiere muchas veces al "number crunching", es decir, a la necesidad voraz e incesante de números: en la preparación de tablas de tiro (ciencia balística), en los complicados cálculos de proyectos científicos (como la "caza" de las partículas subatómicas del premio Nobel Rubbia), técnicos (CAD/CAM) o, más banalmente, en los cálculos contables.

Quizá fueron los trabajos de gestión los que abrieron camino a la idea de que el ordenador podía hacer algo más que una serie de operaciones aritméticas: podía manipular hábilmente muchos otros Tipos de Datos. Cuando se tiene que trabajar con listas (clientes, proveedores, etc.) hay que archivar, leer y manejar cantidades enormes de datos, numéricos y alfanuméricos. Aquí se inició la elaboración no-numérica, que posteriormente ha dado origen a la misma denominación de Ciencia de la Información. Como ocurre siempre, tanto en la historia del hombre como en la de la ciencia, primero se encuentran soluciones parciales, y luego, con

los primeros problemas, surgen momentos de crisis que dan lugar a una más profunda reflexión teórica.

Este proceso se complica por la caótica evolución de los soportes informáticos, favorecida por la ausencia de normas. Enumeramos a continuación algunos soportes que se han ido sucediendo en el campo de la informática: tarjetas perforadas, cintas perforadas, cintas magnéticas, tarjetas magnéticas, tambores magnéticos, discos magnéticos rígidos, flexibles (disquetes o floppy disk). Nos paramos aquí, aunque teniendo en cuenta que, por una parte, hay otros a punto de aparecer (discos ópticos) y que, por otra, cada uno de los soportes mencionados no sólo se basaban en tecnologías diferentes, sino que requerían distintos "formateados" y codificaciones, y, peor aún, una organización y estructura de los datos grabados incompatible casi siempre. Por poner un ejemplo clásico, en la tarjeta Hollerit, adoptada por el IBM, sus 80 caracteres, tantos como columnas agujereadas constituían el registro (o récord): una verdadera camisa de fuerza. Además, la tarjeta de 80 caracteres era "el registro" por antonomasia. Otro límite venía dado en este como en otros sistemas de almacenamiento externos (en la jerga informática se habla de "memorias de masa", para evidenciar su gran capacidad en relación a la de la memoria interna) por el hecho de que el acceso era secuencial, es decir, que para alcanzar un dato (o grabación) intermedio hay que leer antes por fuerza todos los anteriores aunque no nos interesen. Afortunadamente, hoy existen memorias de acceso directo o aleatorio (random en inglés) en las cuales el dato buscado se puede obtener de forma directa, como ocurre en las memorias RAM (siglas que quieren decir Random Access Memory, memoria de acceso aleatorio).

Todos estos problemas iniciales hicieron que se sintiera como una necesidad definir estándares también en los datos, es decir, fijar tipos válidos universalmente. La cosa surgió bastante espontáneamente con el desarrollo de los lenguajes de programación "problem-oriented", dirigidos a los problemas de las personas y no a las exigencias y particularidades de la máquina. Mientras en los lenguajes "machine-oriented" se habla de bits, bytes, campos de tarjetas, etc., en los "problem-oriented" se refieren a los tipos "lógicos" para distinguirlos de los físicos; así existen, por ejemplo, el registro lógico CLIENTE por una parte, y por otra, la posición física en la cual se graban los datos. La grabación física puede diferir del registro lógico en su organización: por ejemplo, en una cinta magnética puede ocurrir que la grabación física contenga en una misma zona los datos de varios clientes, por motivos de economía de espacio; también se pueden encontrar codificaciones que suponen la disgregación de los datos en diferentes localizaciones físicas (Fig. 1). El usuario, ciertamente, no quiere saber

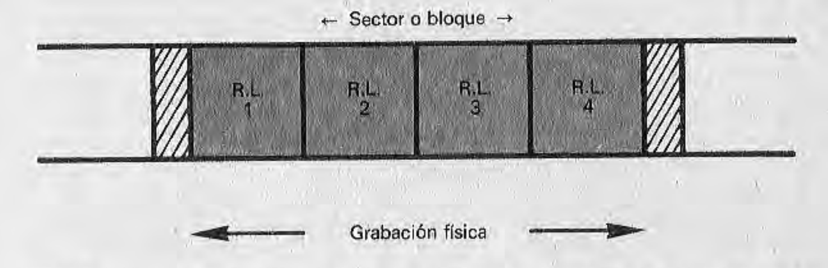


Figura 1.—El registro lógico puede no coincidir con el de la grabación física. Por ejemplo, en una cinta o disco magnético pueden estar contenidos en un mismo bloque o sector varios registros lógicos puestos uno tras otro.

nada de estos asuntos, ni mucho menos enloquecer pasando de un sistema de grabación física a otro. Muy pronto se produjo la evolución de los datos; en un lenguaje de alto nivel no es operativo estar obligado a trabajar sólo con bits y bytes. Así, el vetusto FORTRAN, desde sus primeras versiones ofrecía variables enteras, reales, en simple o doble precisión, alfabéticas, cadenas, y otras más complicadas, como tablas, vectores y matrices.

También en este caso se reafirma el principio de independencia de la máquina, que conduce a la definición lógica o, como hoy se dice, *abstracta* de los datos. Así, pues, habremos de ser abstractos en los datos para que los resultados puedan ser concretos.

Tipos de datos

La programación estructurada —lo hemos dicho más de una vez, pero insistiremos todavía aún a costa de ganar el Oscar del aburrimiento— significa, entre otras cosas, estructuración de los datos. Como se comentó en el apartado precedente, se trata, en primer lugar, de una operación de abstracción, sentido de que prescinde al máximo de las modalidades de implementación de un cierto conjunto de datos. En otros términos, con los *Datos Abstractos*, el énfasis se pone en su significación, en su aspecto lógico. Esto se hace así para asegurar la máxima transportabilidad del software de un sistema al otro. ¿Qué se entiende por "transportabilidad"? Lo explicamos en dos palabras: un software se dice transportable cuando, habiendo sido escrito para un determinado ordenador, funciona a la perfección también en otros.

Este ideal, nacido ya con los primeros lenguajes de alto nivel, culmina hoy con el lenguaje Ada (potentísimo y superestructurado, aunque no adecuado para ordenadores personales), que,

desarrollado para el potente Departamento de Defensa USA, busca ser un verdadero esperanto informático y permitir que el D. D. USA pueda hacer funcionar un determinado programa en una máquina de cualquiera de las innumerables y variadas marcas de las que está dotado el Ministerio de Defensa, con todas sus divisiones y departamentos.

Antes de continuar y de... concretar el discurso sobre la abstracción (¡finalmente!) hay que apuntar, por lo menos, el hecho de que esta tipología abstracta ideal está todavía lejos de ser perfecta. En nuestro tratamiento del Pascal, que tiene fines exclusivamente, didácticos y formativos nos ocuparemos casi exclusivamente de los tipos de datos que residen, también físicamente, en la memoria interna.

El concepto de tipo abstracto de datos significa dos cosas:

- un conjunto de valores o campo de existencia de unas características dadas,
- un conjunto de operaciones que se puedan hacer sobre esos valores.

Un ejemplo banal sería el tipo "entero" (INTEGER en Pascal) que obviamente se refiere a los números enteros algebraicos, y cuyas operaciones son aritméticas; pero podemos añadir otros inventados por nosotros. Veamos un par de ellos, prescindiendo en ambos casos de cualquier lenguaje de programación.

Primer ejemplo: CASAS COLOREADAS.

Definición: el conjunto de casas que hay que pintar

Operaciones: pintar de blanco, pintar de rojo, pintar de azul... (muy a menudo se trata de enumeraciones exhaustivas: ¡se evitarán, espero, casas pintadas de horrendos colores, como el morado o el negro!).

Segundo ejemplo: CARTAS DEL TUTE.

Definición: conjunto de cartas de los cuatro palos (copas, espadas, oros y bastos) del as al rey.

Operaciones: barajar, distribuir a los jugadores...

Ya con el segundo ejemplo se comprende en seguida una particularidad: un tipo, **por ejemplo** CARTAS DEL TUTE, puede subdividirse en sub-tipos (los palos). Para volver a asuntos que tienen que ver más directamente con la informática podemos recordar que un subtipo elemental del tipo "integer" es el CARDINAL, constituido sólo por los enteros positivos (distinción que alguien podría encontrar un poco nimia, pero que a veces es útil). Imaginemos ahora que definimos un nuevo tipo (existe realmente y es muy importante) llamándolo PILA (en el sentido de montón, stack en inglés). Una pila estará formada por datos apilados unos sobre otros, según el orden de llegada y que se pueden extraer uno a

uno de la parte superior (top del stack), evidentemente, en orden inverso respecto al de acumulación (si intentáramos sacar otro situado debajo, los que estuvieran por encima se "caerían"). Por eso el stack se llama también memoria LIFO (Last In First Out, último en entrar, primero en salir), traducción libre del evangélico "los últimos serán los primeros" (Fig. 2).

Las operaciones del tipo PILA que definiremos son, principalmente:

- PUSH (empuja). Consiste en acumular un dato nuevo en la parte superior.
- POP (tira). Operación inversa. Recupera un dato de la cima, con la consiguiente reducción de los elementos de la pila.
- TOP. Copia el dato de la cima, pero sin modificar el contenido de la pila.

Como es obvio, un buen lenguaje de programación debe proveer al usuario de una cantidad adecuada de tipos predefinidos. La potencia de un lenguaje se mide también por la cantidad (y calidad) de los tipos de datos que puede manejar. El pobre BASIC lo es de verdad, bajo este punto de vista, dado que generalmente ofrece sólo los tipos entero, real (en simple y a veces en doble precisión), cadena y matriz.

El aspecto de mayor relevancia de la programación estructurada, presente en todos los lenguajes que se inspiran en ella, reside en que tiene la posibilidad de crear nuevos tipos, definidos por el usuario en base a sus necesidades particulares. Los nuevos tipos se construyen a partir de los predefinidos, ofrecidos por el lenguaje. Una vez más la idea es la de realizar una construcción con las piezas del Lego, o si se prefiere, las de un puzzle.

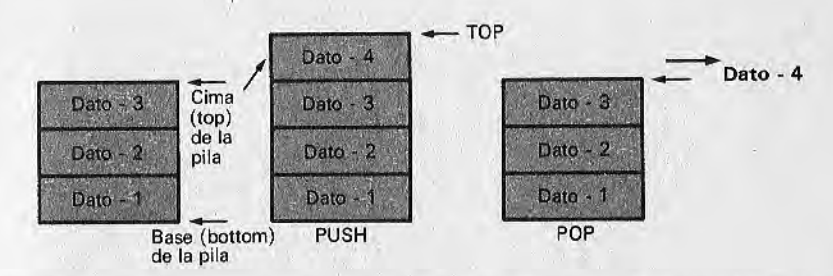


Figura 2.—Esquema general de una pila (stack). La operación PUSH carga un dato nuevo en la cima de la pila, mientras que la POP extrae el dato situado "arriba" modificando la pila. Se puede dar también una operación TOP que coge este dato y lo copia sin modificar la pila.

De este modo se pueden proyectar los tipos más adecuados a cada problema determinando al mismo tiempo una neta y clara separación entre tipos heterogéneos. Dicho familiarmente: no se deben confundir manzanas con peras, tal y como nos enseñaban en la escuela. Las mayores aportaciones a la creación de estos conceptos vinieron del habitual trío de gurús (Hoare, Dahl y Dijkstra). El gran mérito de Niklaus Wirth ha sido implementarlos en sus lenguajes (Pascal y Modula-2).

Pongamos ahora un poco de orden, subdividiendo los tipos de datos en dos categorías:

- simples,
- compuestos.

Tipos simples

Podrían llamarse también atómicos, no por referencia a la energía atómica, sino aludiendo al hecho de que se trata de valores elementales, sin posibles subdivisiones ulteriores. Los compuestos, en cambio, son tipos construidos a partir de los simples, por adición.

Los simples no se limitan a los definidos a priori como parte del lenguaje (típicamente, los conjuntos de números enteros, reales, caracteres alfa numéricos y valores booleanos). El Pascal y otros lenguajes estructurados permiten definir nuevos tipos simples, generalmente mediante dos técnicas: enumeración e intervalo (range).

El ejemplo que sigue aclara estos conceptos: está escrito siguiendo una sintaxis que se asemeja mucho (y a menudo coincide) a la del Pascal, sin atenerse con exactitud a las reglas. En suma, una anticipación que, ya puestos, nos familiariza con el Pascal. Las palabras reservadas o sea, típicas del lenguaje, están escritas todas con mayúsculas.

```
TYPE diasem=(lunes,martes,miercoles,jueves,
             viernes,sabado,domingo);
     dialab=lunes..viernes

VAR x:diasem;y:dialab;
```

Se trata de parte de la sección de un programa en la que se definen los tipos y las variables. Los que conozcan sólo el BASIC notarán en seguida que el Pascal y sus afines son más sofisticados. Por desgracia, es verdad. Como compensación, son extremadamente más claros de leer (¿claridad y síntesis juntas? ¡imposible tenerlas en un mismo lenguaje!). Las líneas que acabamos de ver nos dicen que el programador ha querido definir un TYPE (tipo) diasem, enumerando las siete expresiones alfabéticas entre paréntesis. Después de haber utilizado la técnica de la enumeración ha recurrido a la del rango; creando el tipo dialab, constituido por los días que van desde el lunes hasta el viernes (los cuatro puntitos en la sintaxis del lenguaje suponen todo lo intermedio entre los dos valores señalados). Posteriormente se definen las variables "x" e "y" con la palabra reservada VAR como pertenecientes, respectivamente, a los nuevos tipos de diasem y dialab; la asociación se establece, como pueden ver, mediante los dos puntos (:).

Estamos seguros de que, llegados a este punto, alguien podría encontrar banal todo lo anterior. ¡Y efectivamente, lo es!, pero son precisamente las ideas banales (en apariencia) las interesantes, desde el huevo de Colón a la manzana de Newton.

El mérito de toda esta gente de apellido extraño (Wirth, Dijkstra... ¿cómo se pronuncian?; no lo sabemos) es doble:

- haber trazado un camino, esclareciendo públicamente cosas que quizá algunos programadores hacían ya por instinto;
- poner a nuestra disposición instrumentos prácticos para obtener una implementación cómoda de los tipos más complejos, que nos permitan hacer frente a las más variadas necesidades.

Es inútil decir que en la mayoría de los otros lenguajes tradicionales de alto nivel no podríamos hacer nada por el estilo, y nos las tendríamos que apañar con tipos predefinidos (números o, como mucho, tablas). Con los nuevos medios se gana sobre todo en claridad; además, el lenguaje compilador está ahora en condiciones de señalar, en el momento de la traducción del programa, errores de forma que podrían corresponder a errores de esencia. Pongamos el caso de que, por algún extraño motivo, se asigne a una variable "y", definida del tipo dialab, un valor como sábado o domingo: el compilador, antes de que el programa sea ejecutado, se da cuenta en seguida de que sábado y domingo no pueden atribuirse a una variable que no incluye las fiestas, y nos lo indicaría.

Otra pequeña observación en relación con el fragmento de lenguaje estructurado que acabamos de ver: cuando se define un nuevo tipo por enumeración, como en el caso de diasem, se está dando a la vez una determinada disposición que es la misma de la enumeración. Así, en nuestro ejemplo, el lunes está antes que el martes, que está antes que el miércoles, etc. Esto es lo que nos permite definir después otros tipos por rango, como hemos visto para dialab.

Tipos compuestos o estructurados

Los tipos compuestos (también llamados estructurados) se pueden construir generalmente —es el momento de decirlo— por medio de las siguientes operaciones:

- producto cartesiano,
- unión discriminada,
- array o matriz,
- conjunto potencia,
- secuencia.

Hay que subrayar que se ha apuntado una situación ideal, pues ningún lenguaje es hoy en día tan potente como para implementar todas estas posibilidades, ofreciéndoselas al programador para que se fabrique sus propios tipos. Pasemos lista rápidamente, dando simples ejemplos ilustrativos.

El producto cartesiano parte de dos conjuntos genéricos, A y B, para obtener el conjunto-producto C = A × B cogiendo todas las posibles parejas de elementos, ordenados tomando el primero de A y el segundo de B. He aquí un ejemplo clarificador:

```
TYPE palo=(copas,espadas,oros,bastos);
     valor=1..10
     carta=(p:palo;v:valor)
```

Sin adentrarnos en cuestiones sintácticas (recordemos sólo que los dos puntos, como ya hemos visto, preceden a la indicación del tipo) debería estar claro que el tipo estructurado *carta* está formado por parejas de elementos ordenados, el primero del tipo palo, el segundo de tipo valor. Así, una pareja como (espadas; 1) identifica el as de espadas. Definiendo por enumeración el tipo simple valor, hubiéramos podido tener, obviamente, elementos estructurados como (espadas; sota) (bastos; rey), etc.

Otros ejemplos parecidos podrían ser:

```
TYPE cliente=(no:nombre;dir:direccion;cp:codigopostal;
              ciu:ciudad);
     artic=(cod:codigo;des:descripcion;dp:deposito);
```

Esta estructura corresponde en Pascal, en esencia, al tipo RECORD que viene dado como estándar. He aquí un ejemplo sencillo:


```
TYPE fecha=RECORD
           d:dia;
           m:mes;
           a:anno (* el pascal no acepta la ñ *)
           END;
```

Generalmente RECORD y END son palabras reservadas que también hacen de limitadores, es decir, están en el lugar de los paréntesis de los ejemplos precedentes. Es superfluo decir que se sobreentiende qué nombre, dirección, día, etc., habían sido definidos con anterioridad.

En cuanto a la unión discriminada, parte de dos o más conjuntos para constituir un tercero, en el que están presentes tanto los elementos del primero como los del segundo. Sirve para mezclar los tipos, fundiéndolos en uno de carácter más general en el que, a pesar de todo, cada subtipo conserva su propia estructura, o sea, su identidad. Esta se discrimina o distingue por medio de un oportuno campo común, cuyo contenido es netamente diferente en los dos casos.

Lo explicaremos con un ejemplo, también ilustrado en la figura 3:

```
TYPE autonac=(constr:fabricante;matricula:numatr;
              propiet:persona;primereg:fecha);
     autoext=(constr:fabricante;matricula:numatr;
              nacorig:nacion);
     auto=(nacional:autonac;extranjero:autoext);
```

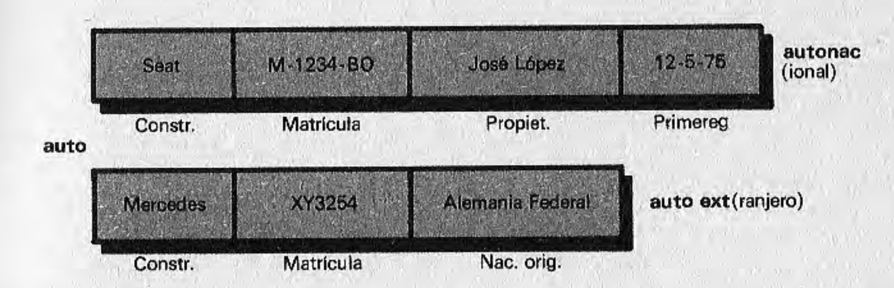


Figura 3.—Con la operación de unión (discriminada) es posible unificar bajo una misma categoría los tipos "con variantes", sobre la base del contenido de un campo discriminante común.

Aquí, los tipos *autonac* y *autoex* (referentes a automóviles nacionales y extranjeros), ya estructurados por su cuenta, tienen en común el campo constr (nombre del constructor), mientras que en el resto tienen diferencias apreciables (el campo nacorig —nacionalidad de origen— que no tiene sentido con coches de tipo autonac). El nuevo tipo *auto* definido por unión (también se dice función OR) permite distinguir del montón aparentemente uniforme los coches nacionales de los extranjeros, en base al campo común constr. Aquí dejamos al lector que imagine cómo se puede pensar en listas separadas de constructores locales o, más drásticamente, en una subdivisión del campo constr en dos subcampos, el primero conteniendo una N o una E, según los casos.

Esta estructura de datos está presente en el Pascal con el nombre de RECORD con variantes.

Veamos ahora los tipos compuestos formados como arrays, ya familiares a los conocedores del FORTRAN o del BASIC. El tipo array (que quiere decir "matriz") es un conjunto *matr* obtenido mediante un conjunto *ind*, de los índices, y otro *val* de los valores. Cada uno de los índices se podrá asociar a uno de los valores. A diferencia de los vectores o tablas de los lenguajes tradicionales, el array, según la concepción de Hoare, permite también la utilización de índices no numéricos.

En Pascal la forma general es la siguiente:

```
TYPE matr=ARRAY[ind] OF VAL;
```

por ejemplo

```
TYPE diames=ARRAY[mes] OF 28..31;
```

el índice *mes* es un tipo no numérico definido anteriormente por enumeración de los meses (de enero a diciembre), mientras que el campo de los valores está definido por intervalos (da sólo los valores posibles, pero no los asocia con el índice; aquí, por ejemplo, no dice que el primer elemento del *mes* tenga 28 días).

Una posible aplicación de este tipo estructurado recién definido podría ser:

```
VAR dias:diames
```

y en el curso del programa se podría utilizar la asignación:

```
dias[febrero]:=28;
```

Para información de los que no lo saben, anticipamos que, en Pascal, el signo = sirve para definir tipos o señalar la similitud entre

términos comparados, mientras que para la asignación se hace preceder el = por dos puntos (:=), así se elimina el equívoco de asignaciones como N = N + 1, (escribiendo N: = N + 1 es más evidente que N se "convierte" en N + 1...)

Otro ejemplo, altamente elocuente (eso esperamos) es:

```
TYPE palopoker=(corazones,diamantes,treboles,picas);
     valpoker=1..3;
     cartapok=(palo:palopoker;valor:valpoker);
     barajpok=ARRAY[1..52] OF cartapok;
```

Veamos ahora el conjunto potencia. Se parte de un conjunto de datos y se forma el de todos los subsistemas que lo componen, incluido el conjunto vacío (a veces llamado "nil" por los informáticos) y el total. Imaginemos un rebaño de cuatro ovejas: a, b, c, d. El rebaño-potencia estaría formado por: el "rebaño vacío" (ninguna oveja), los rebaños de una oveja (a...d), los de dos ovejas (ab, ac, ad, bc..., cd), los de tres ovejas (abc, abd..., bcd) y el abcd. No nos extrañemos de que exista un "rebaño vacío": piense en un pastor al que le han robado todas las ovejas.

En Pascal, la palabra reservada que sirve para definir un conjunto potencia es SET. He aquí un sencillo ejemplo:

```
TYPE colorbase=(rojo,amarillo,azul);
     color SET OF colorbase;
```

después de lo cual es posible, una vez definida una cierta

```
VAR col:color
```

hacer una asignación como col: = (amarillo, azul). Como información diremos que esta estructura tan cómoda en algunos casos es criticada sobre todo por la dificultad de implementaciones eficientes. En efecto, para identificar a cada elemento del conjunto potencia son necesarios al menos tantos bits como elementos tenga el conjunto de que se parte. La identificación más sencilla que se pueda imaginar consiste en poner un bit a 1 ó a 0, según que el elemento esté o no presente (una implementación que utilizase una grabación independiente del resultado exigiría espacios prohibitivos). Por ejemplo, en el caso de la tricromía arriba expuesto son necesarios 5 bits y la asignación col: = (amarillo, azul), correspondería a 011. Imagínese lo que ocurriría con conjuntos base incluso de pocos centenares de elementos.

Aún más pesado desde el punto de vista de la realización es el modo secuencia, que se fabrica con un conjunto X de "n" elementos tomando todas las posibles secuencias (ordendas) de ele-

mentos, desde la secuencia vacía a aquellas formadas por 1, 2..., "n" elementos.

```
TYPE cadena=SEQUENCE caracter
     identificador=(primcar:letra;resto:SEQUENCE(1:letra,d:digito))
```

El primer caso define una cadena como un conjunto cualquiera de caracteres ordenados. En el segundo se trata de un identificador, entendido como la sucesión de letras o cifras, en las que el primer carácter es siempre una letra. Si alguien no ve muy claro todo esto, debe de tener presente que no estamos profundizando en el campo estructurado, con lo cual éstos no son más que sencillos ejemplos (incluso algunas cosas no las profundizaremos ni siquiera posteriormente). En Pascal, el tipo que más se parece a la secuencia es el tipo "file", que constituye una implementación reducida, dado que se refiere solamente a la memoria externa.

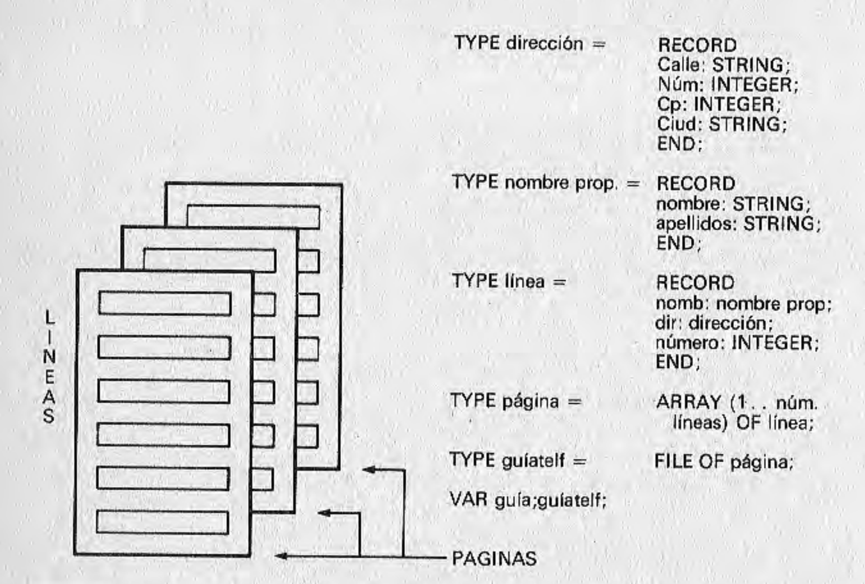


Figura 4.—Con los tipos estructurados es fácil construir con refinamientos sucesivos las más complejas construcciones de datos compuestos. En la figura tenemos el ejemplo de una guía de teléfonos, descrita como un archivo formado por array-página, a su vez compuesto por record-línea, conteniendo cada uno una dirección, subdividida en... Para facilitar su comprensión, la descripción formal de la figura es en bottom-up. De todas formas es evidente el uso del mecanismo nidificado, típico de la programación estructurada.

Después de todo lo hablado sobre los tipos "abstractos" puede parecer una traición, pero el hecho es que también en los terrenos más avanzados se debe distinguir entre sueño y realidad, ideales y límites objetivos.

En la figura 4 hallamos un ejemplo resumido que debería esclarecer el procedimiento top-down con sus fases de refinamientos sucesivos, paso a paso ("step refinements") en la definición de datos estructurados.

Conclusiones provisionales

En la figura 5, para comodidad del lector se representa el cuadro sinóptico de las estructuras presentes en Pascal, anticipando lo que explicaremos mejor más adelante (nos referiremos entonces también a los "pointers" o punteros, que nada tienen que ver con los perros de caza).

Para concretar la panorámica de las prestaciones que se pueden conseguir con la ayuda de Hoare y compañía, traeremos aquí un caso de aprovechamiento de la recursividad ("recursivity" en inglés). Todo proceso recursivo tiene un carácter extraño: se llama a sí mismo. Tendremos ocasión de volver a oír hablar de re-

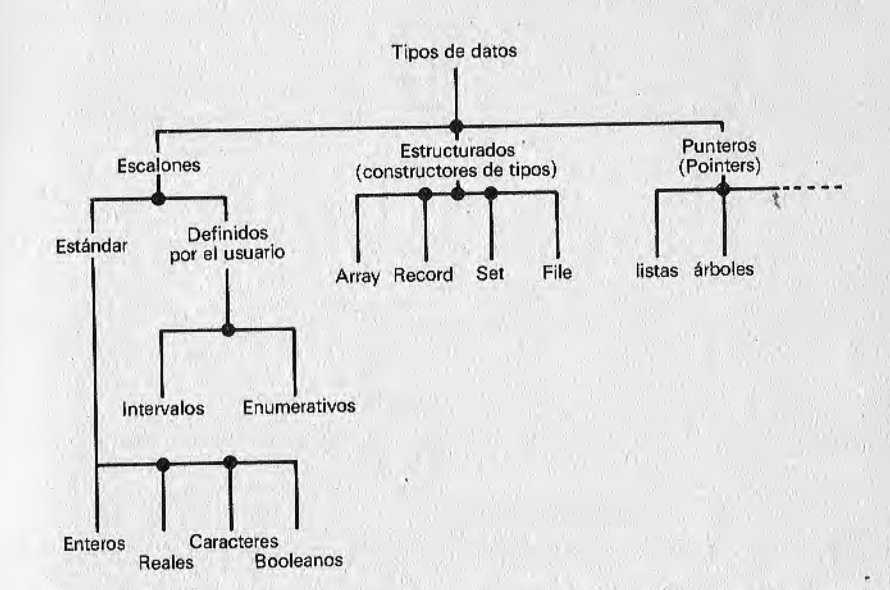


Figura 5.—Cuadro sinóptico de los tipos ofrecidos por el lenguaje Pascal.

cursividad, a propósito de los programas. Este concepto moderno se aplica aquí a la definición de datos estructurados y consiste en el hecho de que la expresión se llama a sí misma desde el interior de su definición:

```
TYPE expresion=SECUENCE termino;
     termino=(addop:operador;f:SEQUENCE factor);
     factor=(muchop:operador;p:primario);
     primario=(CONST:(val:real),VAR:(id:identificador),
              entreparent:(e:expresion));
     operador=(+,-,*,/);
```

(Para ser sinceros, en el Pascal corriente esta estructura "top-down" sería rechazada por el compilador...)

No hay que desesperar si no consigue hacer otra cosa que intuir para qué sirve la compleja construcción precedente (pero debería quedar claro que se trata de una expresión algebraica...) De todas formas hay que explicar que, generalmente, en la notación de Hoare la coma expresa alternativa OR y el punto y coma AND o sucesión. Debe ser suficiente observar que "expresión" figura dentro de su definición.

Para acabar es necesario hacer notar de nuevo que el aprovechamiento de todas las ventajas que se podrían esperar, en teoría, del paralelismo entre la estructura de los datos y los procedimientos de elaboración, no es completa, ni mucho menos perfecta, en los lenguajes estructurados del mundo real. Lo ideal, en efecto, sería que se ofrecieran posibilidades más amplias para definir, junto a los tipos, operaciones características sobre ellos. Quitémonoslo de la cabeza: ni siquiera el tan celebrado Ada (nota 1) cumple en su totalidad esta promesa, que supondría el despliegue de una potencia y una flexibilidad enormes, haciendo posible, entre otras cosas, cambiar las modalidades operativas sin necesidad de modificar las instrucciones.

De todas formas, ahora que estamos a punto de introducirnos en el mundo del Pascal popular, no hay que olvidar un concepto que intentaremos desarrollar ampliamente: Las estructuras de los datos y de los programas están íntimamente ligadas.

Fijemos bien esta frase en nuestro cerebro, porque vale para todos los ambientes de programación, por pobres y espartanos que sean.

Con ello nos adelantamos a una pregunta que se oye a menudo: pero si yo tengo que trabajar en BASIC ¿para qué me sirve estudiar el Pascal si después, en BASIC, no poseo los instrumentos adecuados?

La respuesta tiene tres apartados:

- ¡no es nada malo acostumbrarse a trabajar en Pascal!;
- el Pascal es altamente formativo, aporta ideas de validez general;
- muchas de estas ideas se pueden transportar, con adaptaciones más o menos complicadas, también al BASIC (mejor si es un dialecto moderno, estructurado).

A propósito de las estructuras de datos, en BASIC faltan ciertamente los medios para realizar con comodidad, supongamos, un tipo pila (stack). Es más, el tipo stack siempre será un polizón, subido a bordo sin documentos, o, mejor, escondido bajo un disfraz falso. Nos tendremos que apañar con un vector, lo que conlleva grandes responsabilidades, porque el control del stack y de la legitimidad de sus operaciones tendremos que hacerlo todo nosotros (el intérprete BASIC no nos ayudará en absoluto, pues no sabe nada de pilas ni de cualquier otra cosa que no sea tipo integer, real, string y vector.) La idea de un stack es válida en los casos en que éste sea oportuno y convendrá que pensemos en ello. También con el BASIC.

Nota 1. La ya citada Ada, condesa de Lovelace, era, en vida, la hija del poeta inglés del siglo XIX Lord Byron. Apasionada por las matemáticas, colaboró activamente con el que puede ser definido como el inventor del primer ordenador, con un programa registrado en memoria: Charles Babbage (1792-1871). Por eso, la bella Ada es considerada, aunque con mucha exageración, la primera programadora de la historia. De aquí el nombre dado al lenguaje.

CAPITULO III

LA TORTUGA GUIA NUESTROS PRIMEROS PASOS EN PASCAL

Bottom-up: programación ascendente (o casi)

Después de la panorámica top-down de los capítulos anteriores procederemos ahora justo al contrario con una orientación bottom-up (programación ascendente).

En el desarrollo concreto de los programas es una práctica bastante frecuente que los detalles de un procedimiento (nos tenemos que acostumbrar a este término, que en Pascal designa lo que en otros lenguajes se denomina subprograma o subrutina) sean desarrollados en primer lugar, especialmente si se trata de partes delicadas, que realizan un papel central en el programa y que quizá puedan ser reutilizadas más veces en otro lugar con o sin modificaciones. Ejemplo: un proceso (se dice algoritmo) de ordenación ("sort") o, más banalmente, la búsqueda del mayor elemento en una matriz de datos. Para ligar las piezas del puzzle hay que identificar por lo menos las principales.

¿No contradice todo esto el proceso top-down? En absoluto: una visión de conjunto es indispensable, sólo que en la fase "constructiva" a menudo hace más de "fondo" que de otra cosa. De todas formas se trata de estilo y gustos personales; por lo tanto, dejémonos de discursos y entremos de lleno en la materia. Para hacerlo, adoptaremos la técnica de exposición del profesor Kenneth Bowles, de la universidad californiana de San Diego. Es el padre de la version Pascal UCSD (University of California at San Diego). En realidad, el UCSD Pascal es, más que un lenguaje, un sistema operativo, que integra al Pascal. Por desgracia, los fines que nos

proponemos y el poco espacio disponible no permiten hacer más que una fugacísima alusión a las modalidades con las que en ambiente UCSD se editan y diseñan los programas.

El acercamiento de Bowles es muy útil en plan pedagógico, ya que no requiere de los alumnos ninguna noción previa particular, ni siquiera de matemáticas elementales, por lo menos en un primer momento. En efecto, hace referencia a una tortuga (Turtle), imaginario animalito que está en el centro del lenguaje Logo (como vimos en el volumen 6 de la BBI). Hay que añadir que las instrucciones inherentes a la tortuga no forman parte del lenguaje estándar, sino que se han añadido con fines didácticos, siguiendo la técnica de "librería."

Pero veamos cómo nos podemos familiarizar inicialmente con la tortuga, y así dar también un vistazo al UCSD Pascal. Al principio aparece una línea de órdenes simplificadamente del tipo siguiente:

Command: E(dit, R(un, F(iler, C(ompile, X(ecute).

Es un simple menú interactivo, cuyas opciones se seleccionan tecleando sólo la letra inicial de la orden: E por Edit, F por File, etc. En nuestro caso teclearemos la X de eXecute, después de lo cual escribimos la palabra siguiente:

TURTLE

Es el ábrete-sésamo correcto. En palabras más serias quiere decir que llamamos al programa omónimo de la librería de programas (tiene que estar ya en el disco, si no se producirá un error). Después del habitual zumbido de la unidad de disco (drive), aparece en el centro de la pantalla la flechita que representa el imaginario animalito, que se podrá entonces mover a voluntad. La figura 1 ilustra el efecto de algunas órdenes. Es fácil de comprender, pero de todas formas aprovechamos para recordar que la "Turtle geometry" es una geometría de movimientos relativos. Estos se refieren a la posición actual de la flecha: se puede girar en sentido antihorario (ángulo positivo) u horario (ángulo negativo) o moverla hacia delante (valor positivo) o hacia atrás (valor negativo). En el caso de la figura 1 la sucesión es: al principio la tortuga está en la posición de partida; después se ve el efecto de la orden MOVE (40) que la mueve 40 pasos elementales; TURN (90) la hace girar en ángulo recto; la tortuga se mueve luego 20 pasos. En la quinta figura, en cambio, aparece el efecto de una secuencia de órdenes entre las que se encuentran PENCOLOR (NONE) y PENCOLOR (WHITE), que sirven, respectivamente, para anular y restablecer la acción de la pluma imaginaria asociada a la tortuga;

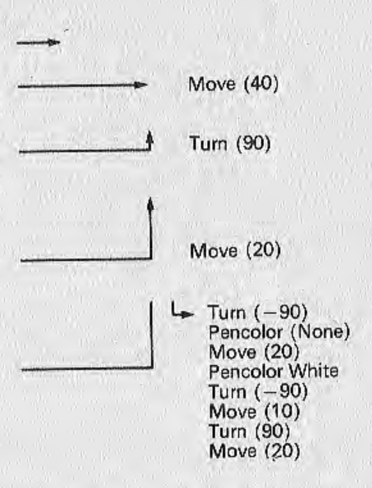


Figura 1.—Movimientos típicos de la tortuga (representada por una pequeña flecha). En el quinto gráfico se tiene el resultado de la serie de órdenes señaladas al lado. PENCOLOR(NONE) da lugar a un desplazamiento sin rastro visible.

en el primer caso, el animalito se moverá sin dejar rastro (con parámetros diferentes a WHITE es posible, con un monitor en color, trazar líneas en colores).

Los números señalados entre paréntesis dependen del ordenador; se expresan en función de la altura y la anchura de la pantalla. Una vez familiarizados, gracias al programa de librería TURTLE, con las órdenes, en ejecución directa, es la ocasión para intentar hacer un programita. Con este fin se debe abandonar TURTLE y volver al menú general de las órdenes, escogiendo esta vez la E de Edit:

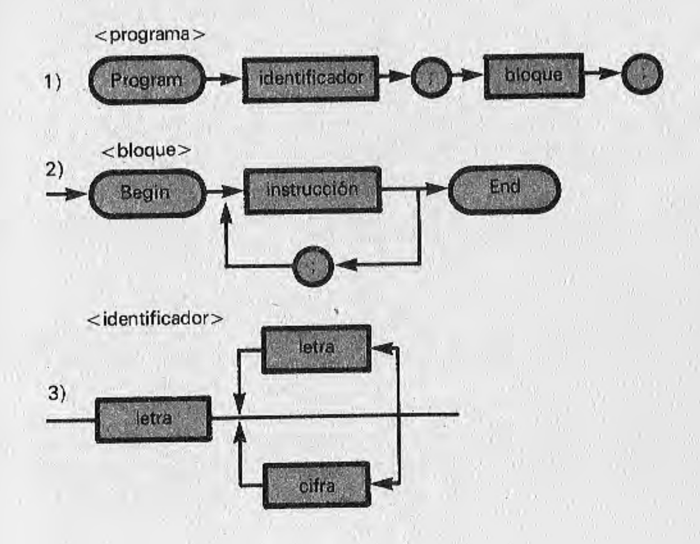
```
PROGRAM primtort;
USES TURTLEGRAPHICS;
BEGIN
 MOVE(50);
 TURN(120);
 MOVE(50);
 TURN(120);
 MOVE(50);
 READLN (* espera return *)
```

Es un programa de una banalidad sobrecogedora, cuya acción, encerrada entre los "delimitadores" BEGIN y END, consiste

en trazar un triángulo de 50 unidades de lado. Pero hemos aprovechado para introducir la sintaxis pascaliana. Diremos, ante todo, que se admiten mayúsculas y minúsculas, pero que nosotros usaremos para distinguir las palabras definidas por el programador las minúsculas (alguna vez con la inicial o alguna letra intermedia mayúscula). Esto ocurrirá con los nombres de programas, procedimientos y datos (o, como se dice en la jerga pascaliana, "identificadores") o bien, con los comentarios (se pueden insertar en cualquier punto, delimitados por una pareja de asteriscos: *comentario*). Dos observaciones: una la instrucción READLN provoca aquí la espera de que se pulse la tecla Return: en este caso específico sirve para mantener la figura en la pantalla, pues de otra forma desaparecería con el END del programa. Segunda observación: la instrucción USES TURTLEGRAPHICS sirve para invocar el correspondiente software de librería (no hay que confundirlo con TURTLE, que es un intérprete de las órdenes dadas a la tortuga desde el teclado), con el cual el compilador completa el estándar del Pascal con las instrucciones de tipo turtle. Por motivos de espacio y sencillez, en los otros ejemplos no insertaremos más USES TURTLEGRAPHICS, aunque en los casos reales siempre hay que ponerlo.

Lamentablemente, a diferencia del cómodo BASIC, el Pascal es un lenguaje no interpretado y semicompilado: la compilación produce un código intermedio llamado "p-code", que a su vez se interpreta en el momento de la ejecución (le aconsejamos olvidar esto por ahora, pues corre el peligro de confundir las ideas). Afortunadamente, el USCD Pascal prevé una especie de atajo. Terminada la edición y reintegrados al menú de las órdenes generales (con la orden Q = "Quit") se podrá escoger la R de R(UN. La primera vez no es como en el BASIC, y efectivamente, aparece un mensaje ("compiling...") que nos invita a tener paciencia. Necesitaremos mucha, especialmente con programas largos, pero si todo está sintácticamente O.K., nuestro programa será lanzado y ejecutado automáticamente. En compensación, una vez compilado nuestro programa (y guardado en disco), sale en seguida y es más veloz que los de BASIC. Así, las complicaciones a las que nos obligan los compiladores (dejamos que se imaginen la dura tarea de eliminar errores, mucho más complicada que en BASIC, con todas las idas y venidas entre Editor y Compilador... ¡pero los hay que se divierten con ésto!) quedan compensadas por beneficios evidentes.

Pero al pretender ser éste un curso de introducción, les remitimos a los manuales para todos estos detalles operativos y pasamos ya a la sintaxis. Para esto haremos uso de la figura 2, en la que rectángulos y círculos expresan las reglas formales del juego, limitadas a las pocas vistas hasta ahora. Se procede con el sis-



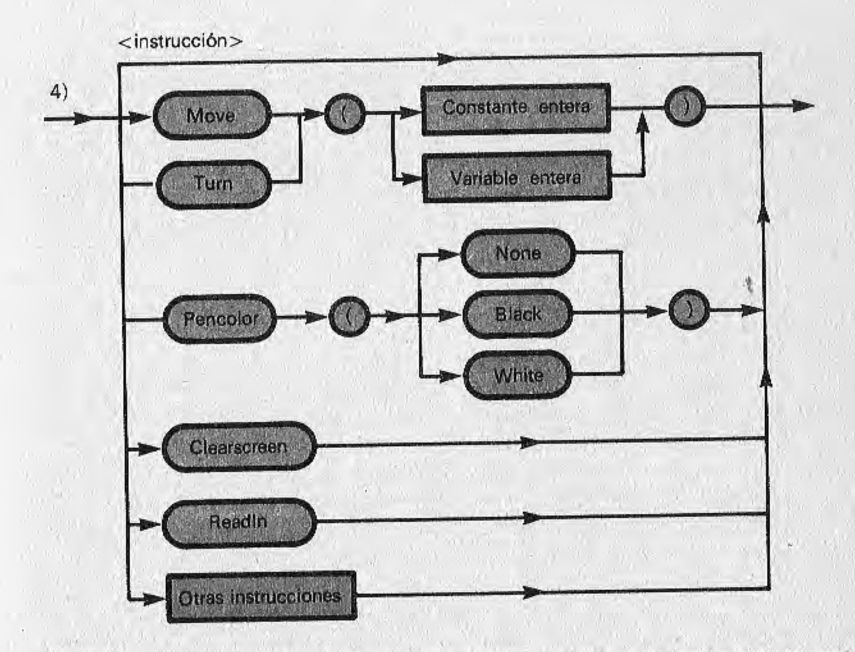


Figura 2.—Cuadro de la nomenclatura y de las órdenes del Pascal vistas hasta ahora. Los símbolos rectangulares encierran cosas a definir posteriormente.

tema top-down; los rectángulos representan cosas aún por precisar en detalle.

En 1) se dice que un <programa> (el nombre de un objeto está encerrado entre paréntesis "angulares"): <programa>, <bloque>, <instrucción>, etc.) está compuesto por la palabra reservada PROGRAM seguida de un <identificador>, un punto y coma, un <bloque> y un punto. Pero, ¿qué es un bloque? Lo aclaramos en 2): en esencia es una serie de instrucciones separadas por ";" entre las palabra BEGIN y END. Este signo de puntuación (;) es importante en Pascal como ya hemos visto en 1).

Pero prosigamos: ¿Qué es un identificador? Una serie de letras y cifras cuyo primer carácter tiene que ser una letra. Añadimos que el número de caracteres significativos es ocho. Con 2) y 3) se entiende fácilmente el significado de las concatenaciones simbólicas: las líneas en paralelo que parten de un nudo expresan alternativa (lógica OR) y una línea sin símbolos significa "posible ausencia" —por ejemplo, en 3) la primera letra del identificador puede ser también la última—. En cambio, aquellas que vuelven atrás denotan posibilidad de repetición o, dicho de una forma culta, "iteración".

En 4) (figura 2) tenemos finalmente el resumen de lo poco visto hasta ahora a propósito de instrucciones, con dos pequeños añadidos como CLEARSCREEN, para el borrado de la pantalla y el rectángulo OTRAS INSTRUCCIONES que nos avisa que todavía quedan muchas otras. En cuanto a los rectángulos que contienen CONSTANTE INTERNA y VARIABLE INTERNA damos por conocidos sus conceptos desde el curso de BASIC (volúmenes 5 y 7); con instrucciones tipo MOVE (lado) o TURN (ángulo) se logra una mayor flexibilidad

Acercándonos a las PROCEDURES

Hagamos aquí algunas observaciones:

- la sintaxis en Pascal es indudablemente un poco complicada, pero la situación es idéntica en todos los lenguajes (el BASIC es sólo más pobre, eso es todo);
- el símbolo de asignación en lugar de "=" común al BASIC y a lenguajes como FORTRAN y COBOL, es ":=" (significa, a grandes rasgos, que la variable del primer miembro "toma" el resultado de la expresión de la derecha), mientras que el "=" queda destinado a las expresiones lógicas;
- las palabras reservadas (las escrias dentro de semielipses en la figura 2, por ejemplo) no se pueden adoptar como identificadores;

- el separador de las instrucciones en Pascal es el ";" únicamente, mientras que el "." decreta el fin del programa;
- la palabra reservada BEGIN es un separador y no va seguida de ";", mientras que la asociada END (tiene que haber exactamente uno por cada BEGIN) requiere el ";" (excepto si es el último del programa, en cuyo caso pondremos "."), aunque, como compensación, la instrucción que le precede no lo necesita (ver dibujo 2 de la figura 2). Idénticas reglas sobre los ";" valen para otras palabras reservadas que funcionan como delimitadores, como FOR, DO, IF, ELSE, etc., pero para simplificar no hablaremos ahora de ellas, invitándoles a seguir los sucesivos ejemplos.

En suma, contrariamente a lo que se cree, en Pascal Return, espacio en blanco y (comentarios) sirven sólo para separar las palabras, y el programa visto (el único hasta ahora) podría muy bien haberse escrito así.

PROGRAM Primtort; BEGIN MOVE (50); TURN (120); MOVE (50); TURN (120); MOVE (50); READLN (* espera return) END.

¡Lo importante es que cada punto y coma esté en su sitio! Por otra parte, los pascalistas nunca hacen semejantes pastiches; es más, recurren a las famosas "indentaciones" que caracterizan el aspecto de los programas Pascal, como veremos. Pero es hora de mostrar algo más serio.

```
PROGRAM stardust;
VAR escala:INTEGER;
PROCEDURE estrella(diamens:INTEGER);
VAR ind:INTEGER;
BEGIN
 TURN(-18);(* centrado estrella en tallo *)
 PENCOLOR(WHITE);
 FOR ind:=1 to 5 do
  BEGIN
   MOVE(DIAMENS);TURN(144);
  END;
 TURN(18) (* tortuga en posicion original *)
END;(* estrella *)
BEGIN (* programa principal *)
 escala:=20;
 TURN(45);MOVE(escala*8);
 estrella(escala*4);
```

```
MOVETO(0,0);TURNTO(165);
MOVE(escala*4);
estrella(escala*8)
MOVETO(0,0);TURNTO(150);
MOVE (escala*6);
estrella(escala*6)
END.
```

El resultado de la ejecución es el trazado de las tres estrellas de la figura 3b.

Veremos en seguida la filosofía de fondo. El programa Stardust se compone de un procedimiento Estrella que es llamado tres veces, cada vez con un valor distinto del parámetro "dimens". Todo parámetro es una variable cuyo valor se fija de forma externa al procedimiento, permitiendole "especializarse", por decirlo de alguna manera. En la sintaxis pascaliana la definición está clara:

```
PROCEDURE estrella(dimens:INTEGER)
```

Esto nos dice que las instrucciones que siguen forman parte del procedimiento de nombre Estrella, que tiene un parámetro, entre paréntesis, llamado "dimens". Los dos puntos que siguen a "dimens" especifican el "tipo" INTEGER (entero) en el caso en cuestión. Las instrucciones de Estrella son las comprendidas entre el BEGIN y el END: un poco más adelante queda claro para qué sirve "dimens": MOVE (dimens), seguido de TURN (144) dice que el lado de la Estrella puede variar mientras que la rotación es fija, e igual a 144 grados. En el programa principal (main) al que, para mayor claridad, se le ha adosado el comentario (*programa principal*), Estrella es invocada tres veces, cada vez con un valor diferente entre paréntesis. Cuando se llama a una PROCEDURE que usa parámetros, en la instrucción de llamada se deberán incluir, en cada caso, los valores que queremos tomen los parámetros en esa ocasión. Así, por ejemplo, la instrucción Estrella (escala*8) "pasa" —realmente se dice así— un valor igual a 8 veces la variable "escala", al parámetro "dimens" de Estrella. Esta es la manera de lograr que el procedimiento Estrella pueda dibujar estrellitas de diferente tamaño (Fig. 3).

Quien sepa BASIC notará la analogía con las GOSUB, salvo que aquí no hay números de línea, los parámetros se pasan desde el exterior y la subrutina se invoca directamente por su nombre: realmente es como si se añadiesen nuevas órdenes a la lista de instrucciones.

a)

PROGRAM Stardust
VAR:....

PROCEDURE Estrella (...
VAR:...

PROCEDURE Hipotet;
VAR:...
BEGIN

END; (*Hipotet*)

BEGIN
TURN (−18)

END; (*Estrella*)

BEGIN (*programa principal*)
Escala:=20

END

b)

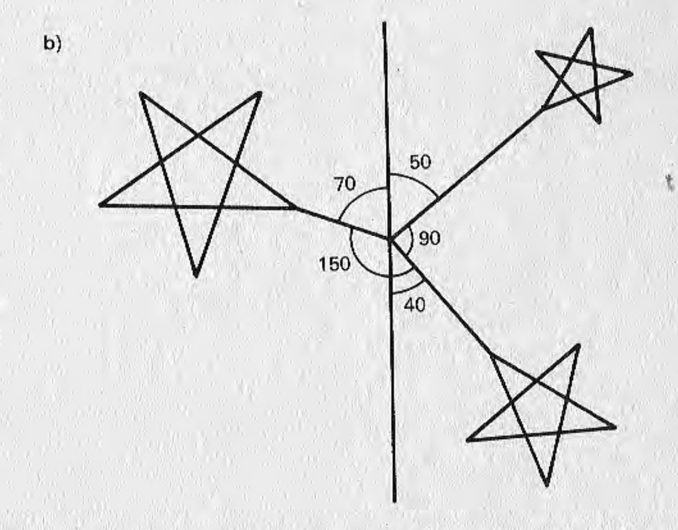


Figura 3.—Estructura anidada de un programa Pascal. En a) está incluido un procedimiento imaginario Hipotet, interno a Estrella. En b) se aprecia un dibujo trazado por el programa Stardust incluido en el texto.

Estructuras de "cajas chinas"

Más que analizar detalladamente lo que hace Stardust, lo que nos urge es comprender su organización. Nos referiremos a la figura 3, donde en a) está representado el armazón de Stardust, vaciada, por así decirlo, de contenidos (como a un cangrejo al que se le ha quitado la carne). Es más, a costa de complicar la vida a los lectores, hemos añadido un tercer procedimiento: Hipotet (o sea, imaginario), "anidado", a su vez, dentro de Estrella. Es un juego de cajas chinas: el programa principal contiene los procedimientos de primer nivel que, por su parte, pueden contener otros de segundo nivel, y así sucesivamente.

Los PROCEDURES forman parte de un concepto más amplio: el de módulos. Con este término se denominan, además de los procedimientos, las funciones definidas por medio de la palabra reservada FUNCTION. Su significado es bastante intuitivo. Los parámetros (entre paréntesis) que emplean las funciones en su definición son llamados también "argumentos". Se trata de nombres familiares por poco que sepan de matemáticas.

Tanto los módulos como el mismo programa global se componen de dos partes:

- declarativa,
- ejecutiva.

Como sus propios nombres indican la primera comprende todas las "declaraciones" necesarias para la correcta interpretación del módulo: nombre, parámetros, tipos y variables, en tanto la segunda se refiere propíamente a las sentencias u órdenes ejecutivas.

Los cuatro elementos posibles mencionados en la primer no están siempre presentes, pero si aparecen deben sucederse exactamente en este orden: nombre (o etiqueta), parámetros, tipos y variables para un módulo. La regla general prevé la sucesión, en el ámbito de un programa completo de Pascal, de las siguientes cosas: nombre, constantes, tipos, variables, procedimientos y funciones. Siempre que se use cualquiera de estos elementos, deberá explicitarse previamente en Pascal. Para mayor sencillez, nos concentramos en las variables. Todas las variables de un módulo, o del programa entero, tienen que ser declaradas, precisando el "tipo" de antemano y no en el momento en que se van a necesitar, como ocurre en el BASIC.

Esto no es una novedad absoluta (también el COBOL prevé una apropiada DATA DIVISION). Las variables que siguen a la palabra reservada VAR están dislocadas meticulosamente en el ámbito de los módulos a los que pertenecen.

Este hecho es típico de la concepción jerárquica, base del estructuralismo. Y no sólo es un hecho formal, sino que afecta al propio desarrollo del programa. En efecto:

- una variable definida en un módulo opera también con todos los de nivel inferior;
- una variable "local" —adjetivo contrapuesto a "global" en esta terminología— sólo opera en el ámbito del módulo en que está definida (y eventualmente en aquellas de menor nivel) y NO deja huella en el exterior.

Volveremos a este punto con un ejemplo concreto. Fíjese ahora en que un módulo que contenga otro de nivel más bajo queda en la práctica dividido en dos por éste. Así sucede con Hipotet, que se interpone entre la parte explicativa de Estrella y la ejecutiva. En particular, esto ocurre casi siempre con el programa principal, cuya parte ejecutiva —tendremos que acostumbrarnos— está al final, precedida por todos los procedimientos y funciones en juego. Indudablemente esto presenta algún problema en el caso de programas muy largos: hay que mirar a fondo el listado para "visualizar" el trabajo completo. De todas formas no prestaremos oído a las provocaciones de quienes, acostumbrados al BASIC, en el que las subrutinas generalmente se escriben al final, realicen comentarios desconsiderados. Nos limitaremos a observar que:

- esta redacción (módulos inferiores precediendo a los superiores) facilita el trabajo del compilador y, los intérpretes, acelera la ejecución (sucede también en BASIC, donde bastaría empezar con un GOTO que se saltara las subrutinas escritas al principio);
- en realidad, el top-down y en bottom-up (programación ascendente) son dos caras del mismo problema.

La esencia de Stardust

Después de todos estos útiles formalismos, volvamos a Stardust para describir lo que hace. Comencemos por la PROCEDURE Estrella. Para entenderlo, basta con meterse en la piel (en este caso caparazón) de la tortuga: después de fijar un ángulo de ataque de -18 y de cargar con tinta blanca (WHITE) su pluma, dibuja los cinco lados de una estrella de lado "dimens". Probemos con el goniómetro o transportador de ángulos: 144 grados es la rotación que hay que realizar para pasar, estando en la punta de una

estrella, del final de un lado al inicio del sucesivo. Para ello se sirve de la construcción Pascal FOR; su sintaxis es:

```
FOR <indice>:=<n1> TO <n2> DO
 BEGIN
 .
 .
 END (* bucle FOR *)
```

En lugar de TO se puede usar DOWNTO, en cuyo caso se contará hacia atrás. Se trata, en esencia, de la misma estructura (FOR/NEXT) existe en BASIC, aunque con dos limitaciones: la cuenta, en el Pascal estándar, sólo puede hacerse con números enteros y el paso (step) es siempre la unidad. Para otras ocasiones quedan las construcciones DO-WHILE y REPEAT UNTIL.

Nos queda ahora el programa principal: es una sucesión normal de desplazamientos de la tortuga y llamadas a Estrella. Entre los primeros aparecen algunos absolutos, mediante las instrucciones MOVETO (x,y) y TURNTO (z), que sirven, respectivamente, para alcanzar el punto de coordenada (x,y) o el ángulo "z" del sistema de referencia de la pantalla, independientemente de la posición actual de la tortuga.

En cuanto a las llamadas del tipo Estrella (escala*8) se aprecia sobre todo la oportunidad de haber introducido la variable global "escala": para referir a una misma unidad de medida, el parámetro "dimens". También es útil la posibilidad ofrecida por el lenguaje de pasar una expresión como valor del parámetro: si hipotéticamente hubiésemos tenido el capricho de trazar una estrella de lado (escala + inc)* 3.14/100, habría bastado escribir:

```
estrella((escala+inc)*3.14(100)
```

Dígalo con flores

El método de la tortuga, además de tener la ventaja de acercar a la informática a gente "pez" en matemáticas, incluso niños, permite hacer entender que un procedimiento no es siempre y necesariamente un conjunto de cálculos, sino también un "hacer algo" (muy útil en robótica, por ejemplo).

Veamos otro sencillo ejemplo:

```
PROGRAM poligonos;
VAR escal:INTEGER;
```



```
PROCEDURE polig(numlad,long,x,y,:INTEGER);
VAR i:INTEGER;
 BEGIN
  MOVETO(x*escal;y*escal);
  PENCOLOR(WHITE);
  FOR i:=1 TO numlad DO
   BEGIN
    MOVE(long*escal);
    TURN(360 DIV numlad)
   END;
  PENCOLOR(NONE)
 END;(* polig *)
BEGIN (* main *)
 escala:=10
 polig(5,16,-40,-40);
 polig(10,8,30,-40);
 polig(30,2,-40,8)
END.
```

No es demasiado difícil darse cuenta de que el procedimiento Polig —que tiene cuatro parámetros: numlad, long, x e y— traza polígonos regulares de numlad lados, cada uno de longitud long, a partir del punto de coordenadas (x*escal, y*escal); lo consigue al trazar lados de valor long seguidos de giros de 360/numlad grados y todo se repite numlad veces, como indica el bucle FOR. Hacemos notar de paso que, para los números de tipo INTEGER (hasta ahora no conocemos otros), el Pascal prevé el operador particular DIV, que proporciona un resultado entero al dividir números también enteros (la división entre números reales se hace con la habitual barra inclinada o "slash").

En cuanto al main, sólo sirve para indicar a Polig los distintos valores de los parámetros. Se obtienen figuras que, al aumentar el número de los lados (y al disminuir long, pues si no se saldrían de la pantalla), se aproximan cada vez más a un círculo.

Vayamos ahora con la siguiente tarea, que consiste en hacer que la tortuga dibuje la planta de la figura 4. Notará en seguida que hay varias coincidencias: los pétalos, se repiten varias veces y también las flores, todas de cinco pétalos, cada uno compuesto por dos arcos. Esto nos sugiere invitar a los más voluntariosos para que escriban el programa ellos solos. Una pequeña ayuda: prueben con procedimientos "anidados", teniendo en cuenta que la flor está hecha de pétalos que están formadas de arcos. Advertimos de todas formas que surgirán problemas algo complicados, aun-

que resolubles. ¿Lo han intentado? Comparen entonces su solución con la siguiente:

```
PROGRAM planta;
VAR escal:INTEGER;
PROCEDURE petalo(long:INTEGER);
PROCEDURE arco;
VAR i:INTEGER;
 BEGIN  (* arco *)
  PENCOLOR(WHITE);
  FOR i:= 1 TO 10 DO
   BEGIN
    MOVE(long*escal);TURN(9)
   END;
 END;(* arco *)
 BEGIN (* petalo *)
  TURN(-45);arco;
  TURN(90);arco;
  TURN(135);
 END;(* petalo *)
PROCEDURE flor(dim:INTEGER);
VAR i:INTEGER;
BEGIN (* flor *)
 FOR i:=1 TO 5 DO
  BEGIN
   MOVE(dim*escal);petalo;
   MOVE(-dim*escal);TURN(72)
  END;
END;(* flor *)
BEGIN (* main *)
 escal:=10;PENCOLOR(WHITE);
 MOVETO(0,-60*escal);
 MOVETO(0,-40*escal);TURN(-45);
 petalo(2);TURN(45);
 MOVETO(0,-20*escal);TURN(45);
 petalo(2);TURN(-45);
 MOVETO(0,0);
 TURN(-45);MOVE(40*escal);
 flor(2);
 MOVE(-40*escal);TURN(45);
 MOVETO(0,20*escal);TURN(45);
```



```
MOVE (20*escal);
flor(1)
END;
```

Los comentarios detallados serían muy largos; le sugerimos encarecidamente de nuevo que estudie con detenimiento todos los pasos que el programa ha impuesto a la tortuga. El procedimiento Arco deriva, imaginariamente, de Polig, aparte del bucle FOR; consiste en 10 segmentos girados entre ellos 9 grados, lo cual, si reflexionamos un poco, quiere decir 90 grados en total, o sea, 1/4 de círculo. Pétalo provocará, por tanto, la siguiente sucesión de ángulos (relativos): -45;90 (las 10 veces 9 grados que acabamos de ver); 90 (antes del segundo Arco) y otros 90 (debidos a la realización del segundo Arco). Esto da un total de 225, a los cuales, antes de salir de Pétalo, se añaden 135 grados, lo que completa el giro, es decir, una vez trazado el pétalo la tortuga queda en la misma situación que cuando lo inició. Todos los módulos siguen el criterio de acabar dejando la tortuga orientada exactamente como en la entrada.

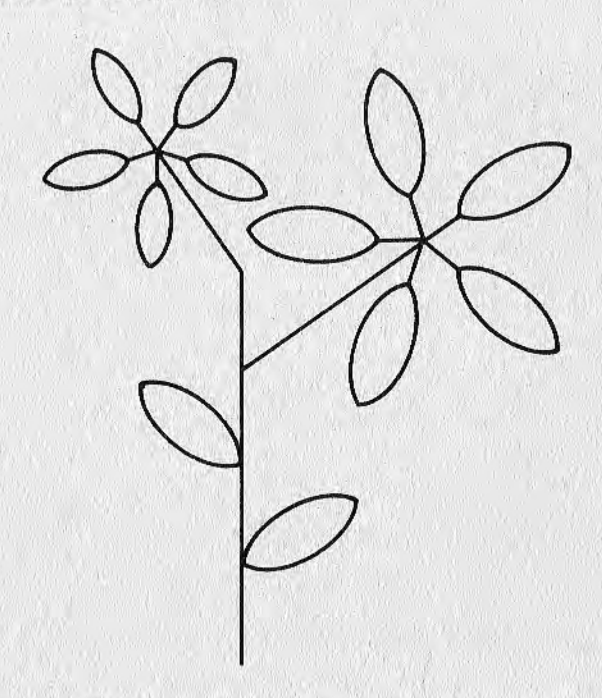


Figura 4.—Dígalo con flores. La programación modular resulta fácil y elocuente con ejemplos tan amenos.

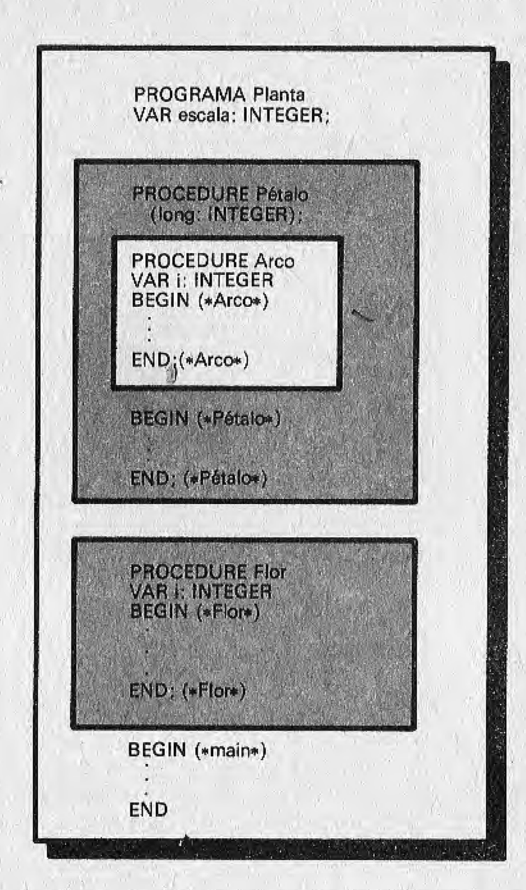


Figura 5.—Estructura de "cajas chinas" del programa Planta. Como se comenta en el texto, no resulta conveniente, como parece a primera vista, incluir Pétalo en el interior de Flor. No todo es coser y cantar...

La figura 5 contiene las "cajas chinas" con la cuales hemos fabricado el programa Planta. A partir de ahora nos las arreglaremos sólo con las indentaciones típicas de los programas Pascal (aunque, repetimos, NO obligatorias), que sirven prácticamente lo mismo para nuestros fines. Queda un asunto en el aire: ¿por qué el anidado de los procedimientos no se ha llevado hasta el extremo que se podría esperar? Y, más exactamente, ¿por qué Pétalo contiene a Arco, pero no se incluye en Flor?

La cuestión es bastante delicada, y para resolverla invitamos a imaginar la solución alternativa (que quizá ya haya hecho alguno de ustedes). La estructura sería del tipo:


```
PROGRAM planta;
VAR escal:INTEGER;

PROCEDURE flor(dim:INTEGER);
VAR i:INTEGER;
PROCEDURE petalo(long:INTEGER);
PROCEDURE arco;
VAR i:INTEGER;
 BEGIN
 ...
 END;
 BEGIN (* petalo *)
 ...
 END;(* petalo *)
BEGIN (* flor *)
...
END;(* FLOR *)
...(seguiria main como antes)...
```

Sin duda se trata de una estructura extremadamente pulcra y elegante y, efectivamente, el profesor Bowles la propone en su libro, pero para trazar una o más flores completas solamente. Nosotros, en cambio, queríamos resolver toda la tarea trazando otros pétalos (en este caso hojas) aislados. ¿Sería lícito, en estas condiciones, invocar Pétalo en el main? La respuesta es negativa. Pueden probar a compilar la nueva versión: el compilador se bloqueará, negándose a proseguir, precisamente al llamar a Pétalo. El motivo está en la estructura jerárquica del proceso de compilación. De aquí se deriva la Regla siguiente:

LOS PROCEDIMIENTOS "VISIBLES" PARA UN CIERTO NIVEL Y LOS UNICOS A LOS QUE PUEDE LLAMAR SON AQUELLOS DE SU MISMO NIVEL O LOS DE NIVEL INMEDIATAMENTE INFERIOR CONTENIDOS EN EL.

En ocasiones, un procedimiento puede invocarse a sí mismo o a procedimientos de nivel mayor..., pero éstos son detalles de los que hablaremos más adelante. Con el último planteamiento visto del main se puede invocar sólo Flor, desde flor sólo Pétalo y desde éste Arco. Nosotros, en cambio, necesitábamos poder llamar a Pétalo en el main, así que lo hemos hecho parejo en "dignidad" a Flor. La solución será, quizá, menos bonita, pero funciona perfectamente (el programa de Bowles sirve para Flor pero no para la Planta entera...).

De esto se puede obtener una pequeña moraleja: aunque el anidar puede ser útil para aumentar la claridad, acercarse al "divide et impera", y en ciertos casos, proteger un procedimiento interno, haciéndolo de uso "exclusivo" de aquel en el que está contenido otro (podría ser el caso de Arco). En general se aconseja mantener sólo dos niveles, el del main y otro más en el que se incluirán todos los demás procedimientos.

Por último, observe que la VARiable "i" está definida en dos procedimientos distintos. Cosas como ésta, que suscitan la curiosidad y que piden explicaciones, las aclararemos en el próximo capítulo (a veces es necesario un poco de suspense).

CAPITULO IV

PROFUNDIZANDO CON LA TORTUGA

Otros formalismos útiles

El programa Planta del capítulo precedente se prestaba a varias reflexiones. Ante todo hay que afirmar que anidar los procedimientos "hasta el final" no es siempre bueno ni, por supuesto, obligatorio.

Muchas veces, sin embargo, este planteamiento se impone por su elegancia; desde luego se corresponde mejor con la filosofía del "divide et impera", que consiste en subdividir mediante niveles jerarquizados, un problema complejo en subproblemas cada vez más sencillos: Planta, que está formada por Flor, que está formada por Pétalo... Hemos aprendido que un procedimiento se convierte en algo privado del procedimiento situado inmediatamente por encima (algo así como las variables locales), con lo que no se le puede llamar desde niveles aún más externos: esto da lugar a algunos problemas.

En cuanto a los datos, hay que distinguir entre globales y locales. De esta sutil distinción dependen aspectos como la seguridad y la comodidad. La seguridad deriva sobre todo del hecho de que se elimina el peligro de que las variables necesarias sólo dentro de un procedimiento puedan ser influidas por el exterior, por despiste o dejadez. Imaginemos, por ejemplo, que hemos escrito en BASIC un subprograma y que queremos reutilizarlo en otro sitio: debemos tener mucho cuidado con los nombres de las variables, pues en BASIC son todas globales. En segundo lugar, y dado que un dato local no deja rastro en el exterior del bloque en el que está definido, o más exactamete, en los bloques de nivel superior, mientras que en general es válido en todos los de

nivel inferior contenidos en el bloque de definición, un mismo identificador puede usarse en procedimientos distintos; así no enloqueceremos buscando nombres de datos.

En el ejemplo del capítulo anterior esto se aplicaría al índice "i", usado tanto en el procedimiento Arco como en el Flor. La variable "i" de Arco y la "i" de Flor son homónimas, pero diferentes, puesto que la primera sólo sería utilizable por Arco y Pétalo (que contiene a Arco), pero no fuera. El compilador les asigna áreas de memoria distintas. Remachemos este punto con un nuevo ejemplo.

```
PROGRAM telescop;
VAR s:STRING;liv:INTEGER;
PROCEDURE primera;
 BEGIN
  liv:=liv+1;
  WRITELN (' ':liv*2,' comienza la primera');
  WRITELN (' ':liv*2,s);
  WRITELN (' ':liv*2,' acaba la primera');
  liv:=liv-1
 END;(* primera *)
PROCEDURE segunda;
VAR s:STRING;
 BEGIN
  liv:=liv+1;
  WRITELN (' ':liv*2,' comienza la segunda');
  s:='¡cucu,soy la segunda!';
  WRITELN (' ':liv*2,s);
  primera;
  WRITELN (' ':liv*2,' acaba la segunda');
  liv:=liv-1;
END;(* segunda *)
PROCEDURE tercera;
 BEGIN
  liv:=liv+1;
  WRITELN (' ':liv*2,' comienza la tercera');
  s:='¡mira que bonito!';
  WRITELN (' ':liv*2,s);
  WRITELN (' ':liv*2,' acaba la tercera');
  liv:=liv-1;
END;(* tercera *)
BEGIN (* main *)
 s:='programa principal';
```



```
  liv:=0
  WRITELN (s);
  primera;WRITELN (s);
  segunda;WRITELN (s);
  tercera;WRITELN (s);
 END.
```

Dado que en este libro, dedicado a los principios básicos, no hay espacio para un tratamiento sistemático completo, vamos explicando ciertas reglas a medida que se van presentando. WRITELN es el equivalente al PRINT del BASIC; concluye con un RETURN. También se utiliza la instrucción WRITE, que funciona de la misma forma, pero sin el Return final. En el Pascal UCSD disponemos del tipo STRING (cadena), ausente en el Pascal estándar. Una constante de cadena se encierra entre apóstrofos (') en lugar de las comillas del BASIC (entre paréntesis). Los objetos a imprimir se sitúan ya sean variables o constantes. Los dos puntos seguidos de una expresión indican la longitud del campo, en el que va justificado por la izquierda el elemento que les precede: así WRITELN(' ':Liv*2, 'comienza la primera') hará preceder la expresión "Comienza la Primera", de tantas parejas de espacios (' ') como le indique Liv*2. Esto servirá para hacer evidente, con la oportuna indentación, el nivel en que nos encontramos y, a tal fin, la variable Liv (nótese que es global) será incrementada y decrementada a la entrada y salida de los tres procedimientos. Llegados a este punto, invitamos al lector a imaginar lo que hace el programa.

Si lo que ha pensado coincide con lo que sigue, ¡felicidades, acertó!:

```
Programa principal
   Comienza la primera
   Acaba la primera
Programa principal
   Comienza la segunda
   ¡Cucú, soy la segunda!
      Comienza la primera
      Programa principal
      Acaba la primera
   Acaba la segunda
Programa principal
   Adelante tercera
   Mira que bonito
   Acaba la tercera
Mira que bonito
```

Seguramente incluso quien haya seguido paso a paso y escrupulosamente el programa, se habrá encontrado con alguna sorpresa. La razón de esta extraña apariencia reside en el hecho de que la "s" es global y actúa en el main como en todos los procedimientos excepto en Segunda, en donde tenemos con el mismo nombre una variable local. Por lo tanto, la cadena global "s" permanece inactiva (ignorada y "en suspensión") dentro de Segunda, y vuelve a funcionar en cuanto se entra en un bloque en el que "s" no es local. Se puede ver incluso que cuando llamamos desde Segunda a Primera el valor de "s" que se imprime no es el esperado "¡Cucú, soy la segunda", sino "Programa principal", y ocurre esto porque en Primera "s" vuelve a ser la global, desactivada tan sólo en Segunda.

Es muy importante dominar por completo este funcionamiento: corresponde a un mecanismo preciso del compilador que, cuando llega a una instrucción, busca la variable en primer lugar en la lista de las variables locales; si encuentra la que está en juego en la instrucción, se da por satisfecho, y sólo en caso contrario mira en los módulos de nivel superior a los que pertenece (no en los demás) y, en último extremo, en la lista de variables globales. Espero que ya esté todo claro, especialmente el por qué con la última instrucción del main se escribe 'Mira qué bonito' (el procedimiento Tercera no tiene "s" como variable local y, por tanto, actúa sobre la "s" global.

La ocasión es propicia para prevenir posibles críticas por parte de los BASICalistas empedernidos, invitando, sobre todo a los principiantes, a no exagerar con los refinamientos de la programación en Pascal, para evitar malentendidos. En concreto, esto se puede "traducir" como:

- no usar demasiados niveles, adoptándolos sólo allí donde resulten lógicos y funcionales;
- distinguir al máximo (y en caso de duda conviene recurrir a nombres distintos...) entre variables locales y globales.

Otros elementos del mosaico: las construcciones

Ha llegado el momento de tratar de las estructuras de control (releer los capítulos 1 y 2 puede ser oportuno).

En la figura 1 están dibujados los elementos sintácticos del ciclo FOR (ya visto) y de los famosos WHILE y REPEAT. Como se puede apreciar (diagrama 4), por Instrucción Compuesta se entiende una serie de instrucciones elementales (llamadas en inglés "statement") separadas por el habitual ";" y delimitadas por los familiares BEGIN y END. La estructura REPEAT, en cambio, prescin-

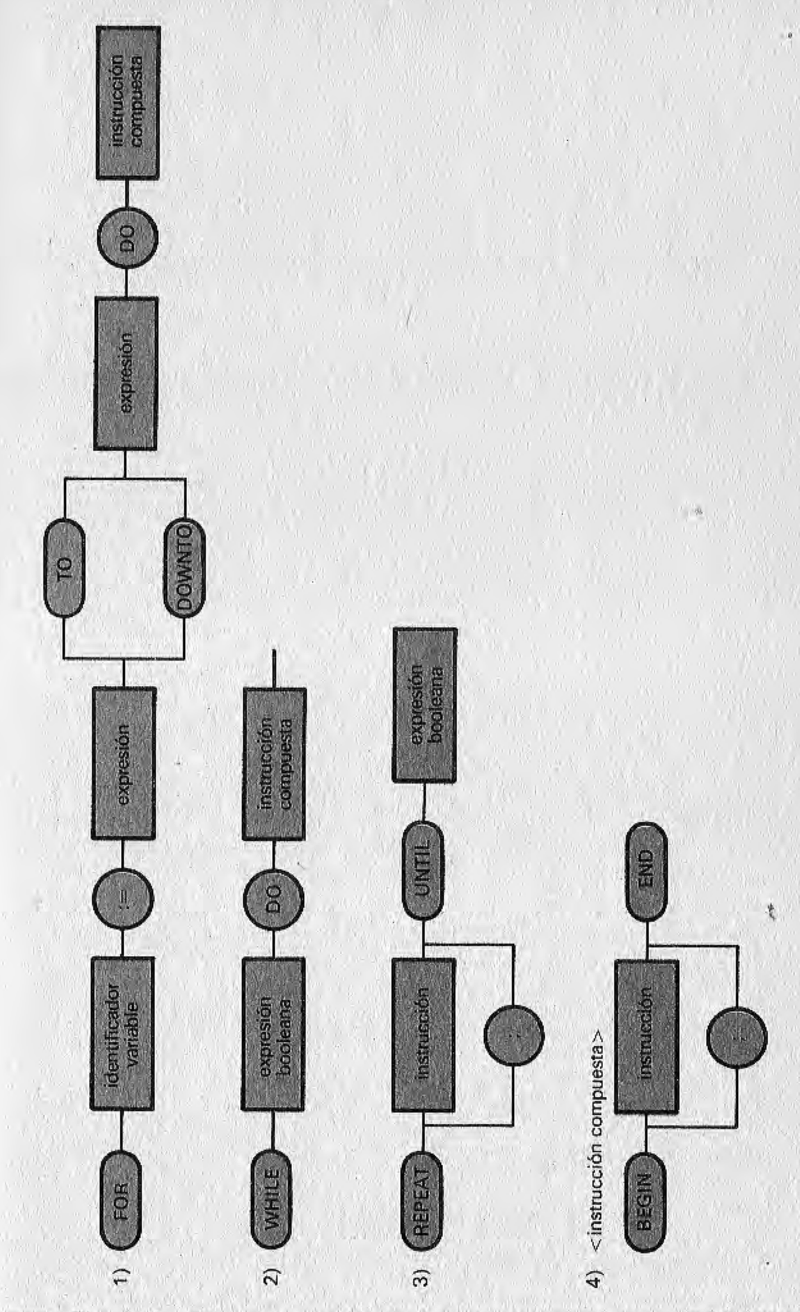


Figura 1.—Elementos sintácticos de las construcciones FOR WHILE y REPEAT.

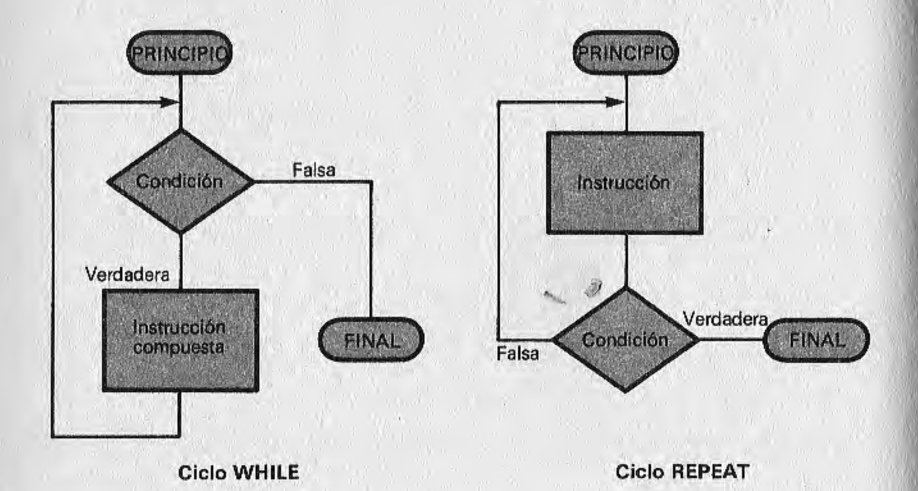


Figura 2.—Diagrama de flujo de las estructuras cíclicas WHILE y REPEAT en Pascal. Recuerden que con el primero se sale cuando la <condición> es falsa y, con el segundo, cuando la <condición> es verdadera.

de de ellos (esto pasa en Pascal; en otros lenguajes estructurados hay mayor uniformidad al respecto). En efecto, el padre del Pascal, Niklaus Wirth, decidió que las palabras REPEAT y UNTIL constituían buenos delimitadores "naturales".

En la figura 2 tenemos los organigramas de las dos construcciones WHILE y REPEAT. Son equivalentes y siempre se pueden sustituir una por otra.

Ha aparecido en la figura 1 un término (expresión booleana) que es oportuno aclarar. En la jerga se llama booleana a un tipo estándar de variable capaz de tomar sólo dos valores: TRUE (verdadero) y FALSE (falso), palabras evidentemente reservadísimas. El adjetivo "booleano" se usa en honor a George Boole, inventor del Algebra del mismo nombre. Este adjetivo caracteriza en Pascal una variable lógica, de la siguiente forma:

```
VAR lapalis:BOOLEAN;
```

y después de esto, una posible asignación de esta variable, sería:

```
lapalis:=(a>b) OR NOT (a>b)
```

Pregunta, ¿qué valor (booleano) asumirá lapalis según la condición anterior? Es evidente que TRUE, independientemente de

los valores asumidos por a y b; o es cierta una condición o la otra, no hay escapatoria. Las "reglas del juego" de la expresión booleana, en Pascal, son las mismas que en el BASIC, es decir:

- las condiciones elementales están expresadas por comparaciones (con simbología idéntica: >,<,=,>=,<=,<>);

Las comparaciones deben hacerse entre tipos homogéneos:

- el símbolo = en Pascal no se puede confundir con el de asignación, que, como se ha visto, está precedido por dos puntos;
- se pueden usar los operadores lógicos AND, OR y NOT.

Veamos ahora un par de ejemplos con la tortuga y, en parte, con las cadenas del UCSD Pascal. Este es el primero:

```
PROGRAM whilegrf;
VAR long,ang,incr:INTEGER;
    resp:CHAR;
PROCEDURE iniscrn;
 BEGIN
  CLEARSCREEN;PENCOLOR(WHITE)
 END;
PROCEDURE proxlin;
BEGIN
 MOVE(long);TURN(ang);
 long:=long+incr
END;
BEGIN (*main *)
 iniscrn;
 WHILE (resp<>'F') AND (resp<>'f') DO
  READLN (ang);READLN(incr);
  long:=5;
  WHILE long<=150 DO proxlin;
 READ (resp);
 iniscrn;
END; (* WHILE mayor *)
END.
```

Este programa sirve para trazar un típico dibujo de los gráficos de Tortuga, como el de la figura 3 (corresponde a un ángulo = 89 grados). Variando los parámetros "ang" e "incr" introducidos

se podrán ver infinidad de resultados (son interesantes ángulos como: 59, 60, 61, 72, 73, 119, 120, 121, 135, 144, 154, etc.). Quien disponga de un ordenador y se quiera divertir, puede hacer lo siguiente: ante todo, insertar después de la línea PROGRAM el indispensable USES TURTLEGRAPHICS y sustituir CLEARSCREEN por INITTURTLE. El main está compuesto por dos ciclos WHILE anidados uno en otro. El exterior toma los parámetros "ang" e "incr" y, al final, "resp". Este es un dato de tipo CHAR, es decir, carácter, por lo que el READ equivale al GET de BASIC. Como resulta evidente (y es ésta la razón tanto de WHILE como de REPEAT) el proceso se repite "mientras" (while) la respuesta ("resp") sea distinta de fin (F mayúscula o minúscula). Una pregunta: la condición (resp <> 'F') AND (resp <> 'f') es correcta, pero sustituyendo AND por OR, el ciclo no acaba nunca. ¿Sabría decirnos por qué? Si no se pulsa F ni f (un simple RETURN, por ejemplo) se llama al procedimiento Iniscr de limpieza de pantalla. En cambio, el bucle más interno ejecuta repetidamente Proxlin, que consiste en trazar líneas de longitud creciente (en un valor incr) seguidas de un giro de "ang" grados hasta que "long" supere 150.

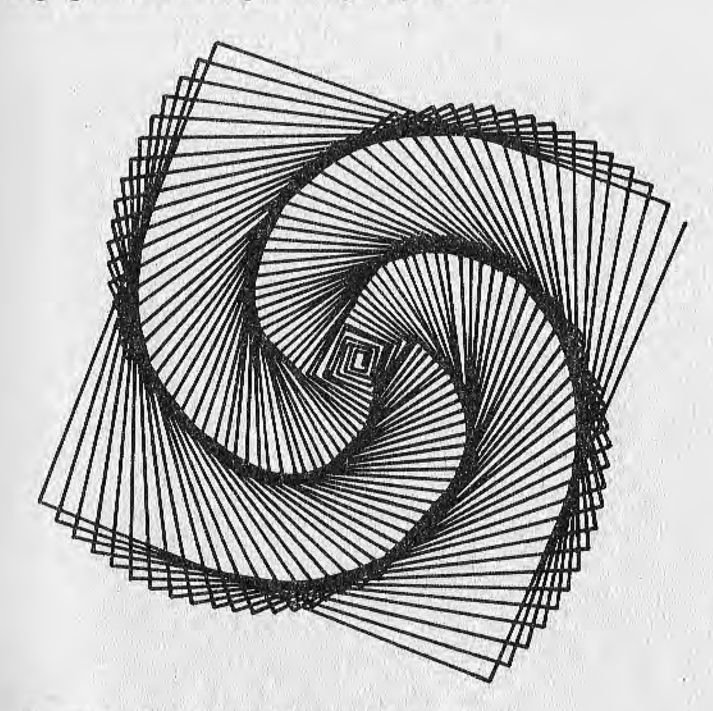


Figura 3.—Un clásico dibujo de la tortuga.

Tome nota de que el WHILE más interno no ha requerido los delimitadores BEGIN y END porque, como suele ocurrir en otros lenguajes, comprende una sola instrucción: Proxlin.

Una última observación: lo espartano del diálogo hombre-máquina es también debido al hecho de que, introduciendo la tortuga, se va a página gráfica, por lo que instrucciones tipo WRITELN ("Dame el ángulo") no aparecerían en la pantalla de muchos ordenadores personales,, a menos que no se introdujeran instrucciones de restablecimiento de modo texto/modo gráfico (TEXT MODE y GRAFMODE en el Pascal UCSD del Apple) que complicarían innecesariamente el ejemplo.

REPEAT, IF THEN/ELSE y equivalencias

Al ilustrar ahora la construcción REPEAT, aprovecharemos también para introducir la estructura IF/ELSE, muy conocida y presente en muchos BASIC.

La idea original del algoritmo base del siguiente programa se remonta al célebre Martin Gardner, de la revista "Scientific American", y consiste en transformar una cadena de caracteres 'S' y 'D' en figuras obtenidas haciendo girar la Tortuga +90 ó -90 grados en cada uno de los casos. Para complicar las cosas hemos añadido la posibilidad de que el usuario pida la repetición del trazado y de que elija ángulos diferentes de 90. Los dibujos obtenidos son del estilo de los mostrados en la figura 4: en a) con la limitación de Gardner y en b) con la flexible extensión a ángulos cualesquiera.

```
PROGRAM figuras;
VAR sec:STRING;car:CHAR;
    dim,ang,i:INTEGER;
BEGIN
 WRITE ('¿dimension? ');READLN (dim);
 WRITE ('¿angulo? ');READLN (ang);
 WRITE ('¿secuencia? ');READLN (sec);
 CLEARSCREEN;PENCOLOR(WHITE);
 REPEAT
  i:=1;
  REPEAT
   MOVE(dim*i);
   IF SEC(i)='S' THEN TURN(ang)
      ELSE TURN(-ang)
   i:=i+1;
```

```
    UNTIL i>LENGTH(sec);
    READ(car) (* NOTA: Return acaba la sesion de dibujo *)
  UNTIL car=CHR(13)
END.
```

Se trata de uno de esos casos en los que el programa habla por sí mismo, es casi autoexplicativo: hasta que (until) el usuario no apriete la tecla Return (de código ASCII 13) la tortuga repite el trazado (recomenzando con i:=1) de la figura (Gardner utilizaba el término "spirolateral"). Si no acaba con la tortuga en la misma posición que al inicio, cada una de las repeticiones se convertirá, en realidad, en fragmento de otra figura mayor. La figura "nace" de la interpretación de la cadena *sec*, cuyos caracteres son examinados, uno a uno, con la instrucción sec(i), determinando giros a derecha o izquierda y repitiéndolos hasta...; ¡bueno, no vamos a repetir lo que, en el fondo, se dice y repite en el programa mismo! Haremos sólo unas observaciones formales sobre los elementos pascalianos introducidos a hurtadillas:

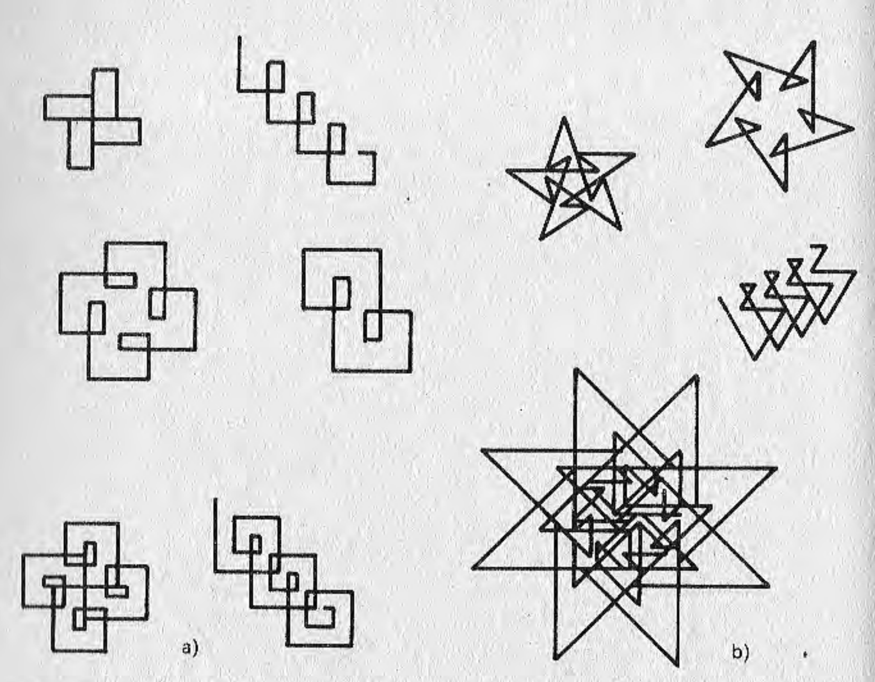


Figura 4.—Figuras obtenidas partiendo de un algoritmo ideado por Martin Gardner. En a) se sigue su idea primitiva (ángulos de ± 90 grados); en b) se representan ejemplos de la variante de Bowles (cualquier tipo de ángulo).

- el tipo CHAR es, obviamente, un carácter;
- WRITE('algo') seguido de READLN(x) equivale a INPUT "algo", x del BASIC, y WRITE, contrariamente a WRITELN, no manda a principio de línea el cursor;
- sec(i) funciona en una cadena como el MID$(SEC$,I,1) dándonos el iésimo carácter de la cadena;
- también LENGTH (cadena) y CHR(n) funcionan del mismo modo que las funciones análogas en BASIC dan la longitud de la cadena y el carácter correspondiente al código ASCII n.

Y veamos ahora el problema de la equivalencia entre REPEAT y WHILE, que ya sabemos garantizada por el teorema de Jacopini & socio.

En casos como los dos que mostramos a continuación (generación de los cuadrados de 1 a 100...) es banal:

```
x:=0;                   x:=0;
WHILE y<100 DO          REPEAT
BEGIN                   x:=x+1;
x:=x+1;                 y:=x*x;
y:=x*x;                 ....
....                    UNTIL y>100
END;
```

Observe que la condición y<100 del WHILE se ha cambiado por y>=100 en el REPEAT. Esto sucede porque el test se hace en un caso al principio y en otro al final, y por la semántica diferente de WHILE y REPEAT (vuelvan a estudiar la figura 2).

Pueden darse, sin embargo, situaciones de relativa ambigüedad.

Supongamos que tenemos:

```
READLN(x,y)
WHILE x>=y DO
 BEGIN
  elab(x,y)
  ....
 END;
```

donde Elab(x,y) sea un procedimiento que manipula "x" e "y", transformándolos de forma imprevisible (Elab podría contener,

por ejemplo, la lectura de una nueva pareja x,y), de forma que no sabemos *a priori* cuando se conseguirá la condición prevista. Ahora bien, el ciclo WHILE, con el test al principio, podría no realizarse ni siquiera una vez; así, una trasposición precipitada al ciclo REPEAT, que tiene el test al final, podría resultar errónea. La correcta será la siguiente:

```
READLN(x,y)
IF x>=y THEN REPEAT
  elab(x,y);
  ....
  UNTIL X<Y
```

No hace falta, empero, ser muy puristas para encontrar esta solución poco elegante. Veamos un ejemplo totalmente formal del caso contrario:

```
REPEAT
I1;I2;...;In
UNTIL <CONDIC>
```

Donde I1...Im son instrucciones genéricas.

Para hacer algo perfectamente equivalente con WHILE tendremos:

```
I1;I2;...;In
WHILE <CONDIC> DO
BEGIN
I1;I2;...;In
END;
```

Nótese que <condic'> es, generalmente, la negación de <condic>. Ahora bien, si las instrucciones I1...In forman un conjunto abundante (con bastante código), su duplicación resultará, por lo menos, antieconómica. Se podría evitar adoptando una variable auxiliar booleana (como puede ver, volvemos a la problemática del capítulo 2):

```
sw:=TRUE;
WHILE <CONDIC> OR sw DO
BEGIN
I1;I2;...;In;
sw:=FALSE
END;
```


En el primer ciclo, el switch sw (desviador), al ser Verdadero, fuerza la primera ejecución. Al acabar ésta, le damos a SW el valor FALSE (falso), con lo cual, que se realicen o no más ciclos dependerá sólo de <condic>.

Está claro que, según las ocasiones, unas veces REPEAT será más cómodo que WHILE y otras ocurrirá al contrario. También la presencia del bucle FOR es útil, aunque las limitaciones ya vistas nos obliguen, a veces, a recurrir a WHILE o REPEAT para lograr un incremento/decremento no unitario. Hay también algunos trucos para lograr esto sin usar las mencionadas estructuras: por ejemplo, un equivalente a un bucle BASIC de tipo:

```
FOR X=0.1 TO 10 STEP 0.1
```

se puede hacer en Pascal así:

```
x:=0
FOR i:=1 TO 100 DO
 BEGIN
 x:=x+0.1
 ....
 END;
```

Funciones, procedimientos y sus relaciones

Como ya comentamos previamente, en Pascal, además de los procedimientos, existen las FUNCTION. Su finalidad, a diferencia de los primeros, es restituir al programa (o subprograma) que las llama un valor evocable a través del nombre de la propia función.

Como de costumbre, vamos a dar en seguida un par de ejemplos:

```
PROGRAM cilindro;
VAR volumen,r,h:REAL;
FUNCTION areacirc(r:REAL):REAL;
CONST pi=3.14159265;
 BEGIN
  areacirc:=r*r*pi
 END;
BEGIN (* main *)
 READLN (r);
 WHILE r>=0 DO
```

```
BEGIN
 READLN(h);
 volumen:=areacirc(r)*h;
 WRITE ('el cilindro de radio ',r,' y altura ',h);
 WRITELN ('tiene un volumen ',volumen);
 READLN(r)
END;
END.
```

Hemos aprovechado el ejemplo para introducir otro tipo de dato, el REAL, cuyas características pueden encontrar en anteriores volúmenes de la BBI (son números reales, por ejemplo, 10, 3.14, 8.1697, etc.). También hemos introducido otro elemento sintáctico: CONST, que declara como constante el nombre que le sigue (en nuestro caso "pi") y fija su valor (3.14159265), "pi" será desconocida fuera de la FUNCTION Areacirc. Dese cuenta de que se utiliza el signo = en lugar de := por cuanto se tiene identidad pero no asignación.

Este ejemplo es tan sencillo que su funcionamiento para el cálculo de volúmenes de cilindros se deja enteramente a la comprensión del lector. En esencia nos ha servido para introducir la sintaxis de FUNCTION y para hacer notar que en ella, a diferencia de lo que sucede con un procedimiento, el nombre de la función se conduce, en una expresión, como una variable, que puede utilizar de forma directa el programa que la llama. Esto podría incluso realizarse con un WRITELN; en el caso visto, se habría podido hacer lo siguiente:

```
WRITE ('el cilindro de radio ',r,' y altura ',h);
WRITELN ('tiene un volumen ',areacirc*h);
```

En otro caso, suponiendo que Prisma, Pirámide y Esfera sean otras FUNCTION, el volumen total de un compuesto sólido sería dado por una expresión de este tipo:

```
voltot:=prisma(b1,b2,h1)+piramide(b3,b4,h2)+esfera(r)
```

formado por ellas (admitiendo que la esfera se sostuviera en la punta de la pirámide).

Muchas veces es posible obtener con los PROCEDURES algo muy similar a FUNCTION. No es un juego de palabras; por ejemplo, en nuestro caso hubiéramos podido recurrir a la variante siguiente:

```
PROCEDURE calcarea(rag:REAL;VAR areacirc:REAL)
```


Pero cuidado: después hay que modificar también la declaración de las VAR en el main. Se puede hacer así:

```
PROGRAM cilindro;
VAR volumen,areacirc,r,h:REAL;
```

De esta forma logramos que Calcarea devuelva, dependiendo sólo del parámetro rad el resultado Areacirc. Entonces, ¿queda todo lo demás como antes? No exactamente. Tal y como dijimos antes, de esta forma podemos lograr "algo" parecido a una FUNCTION, pero su comportamiento y manejo serán distintos. Así será necesaria también una modificación posterior.

```
....
READLN(h)
calcarea(r,areacirc);volumen:=areacirc*h;
```

Seguramente la mayoría de ustedes ya se esperaban algo parecido, por lo menos los más atentos, pues recordarán que un procedimiento debe ser llamado explícitamente, con sus parámetros, para ser ejecutado. Lo que sí puede haberles pillado de sorpresa es la presencia de Areacirc junto al parámetro "r". En efecto, Areacirc es, al mismo tiempo, un parámetro y un resultado. Para poner un poco de orden en esta delicada materia debemos aclarar que, en los procedimientos, hay dos tipos posibles de parámetros:

- parámetros llamados "como valor",
- parámetros llamados "como referencia" (o "por situación").

También se les llama, respectivamente, parámetros-valor y parámetros-variables.

Los primeros son como rad, y los segundos son del tipo de Areacirc y se distinguen de los primeros por estar precedidos en su definición de la palabra VAR. La diferencia esencial entre ellos es que los primeros sólo son argumentos, mientras que los segundos pueden ser también funciones.

Pero ambos son parámetros y, por lo tanto:

- hay que pasarlos al procedimiento,
- pueden tener nombres distintos en el main y en el procedimiento.

Para entendernos mejor, supongamos que el ejemplo precedente sea un poco más complicado y se tengan que calcular los volúmenes de tres cilindros distintos de radios r1, r2 y r3, y alturas h1, h2 y h3. Incluimos en las VAR del main no sólo estas seis nue-

vas variables, sino también area1, area2 y area3. Intente, sabiendo esto, escribir la llamada correcta a Calcarea para calcular el volumen del segundo cilindro (suponemos que "volumen" es una variable de trabajo usada para los tres cilindros). Sería:

```
calcarea(r2,area2);volumen:=area2*h2;
```

Es decir, que la elección de Areacirc como nombre común para el main y el procedimiento era legítima (no producía molestias), pero, ciertamente, no vinculante.

Veamos un caso simple, pero interesante; para aclarar otros cuantos conceptos:

```
PROGRAM ejemplo;
VAR x,y,z:INTEGER
PROCEDURE contador(VAR n:INTEGER;incr:INTEGER);
  BEGIN
    n:=n+incr
  END;
BEGIN (* main *)
 x:=0;y:=2;z:=1;
 WHILE x<12 DO
   BEGIN
     contador(x,z);
     contador(y,x);
     contador(z,y);
   END;
END.
```

¿Después de cuántas vueltas se sale del bucle WHILE y con qué valores de "x", "y", y "z"? Si lo hace verá que tras dos vueltas y con valores 17, 25 y 37 (basta un poco de paciencia para comprobarlo). A mitad de camino puede suceder, por ejemplo, que "x", "y" y "z" valgan 1, 3 y 4, después de lo cual, al llamar a Contador (x,z) a "x" se le añade "z", "pasada" como incr a Contador, que devuelve una "x" modificada igual a 1 + z = 5.

En resumen: en una PROCEDURE los parámetros usados como referencia son empleados para facilitar los resultados obtenidos en la PROCEDURE al programa o subprograma que realizó la llamada y que determinó los parámetros de valor.

Otra ventaja de los procedimientos respecto de las funciones es que permiten la realización de funciones con salidas múltiples. Sólo será necesario definir más parámetros de tipo VAR. Por ejemplo (y sin más comentarios):


```
PROCEDURE multfunc(x,y,z:REAL;VAR f1,f2:REAL);
```

Hay una última diferencia entre FUNCTION y PROCEDURE, pero es un tema delicado que lo dejaremos para más tarde.

Otras aplicaciones de FUNCTION e IF THEN/ELSE anidadas

A pesar de lo anterior, muchas veces se pueden obtener cosas elegantes con una FUNCTION, además de instructivas. He aquí una:

```
PROGRAM preguntas;
VAR trabajo,lluvia:BOOLEAN;
 FUNCTION si:BOOLEAN;
 VAR resp:CHAR;
 BEGIN
  READLN(resp);
  IF resp='S' OR resp='s' THEN
   si:=TRUE
  ELSE
   si:=FALSE
 END;(* si *)
BEGIN (* main *)
 WRITELN ('¡despierta!');WRITELN;
 WRITE ('¿casado desde hace mas de 5 años? ');
 IF NOT si THEN WRITELN ('besa a tu mujercita');
 WRITE ('¿dia laborable? ');trabajo:=si;
 WRITE ('¿llueve fuera? ');lluvia:=si;
(* Se podrian añadir muchas mas preguntas *)
 IF trabajo THEN
   IF lluvia THEN
     WRITELN ('al trabajo en coche')
   ELSE
     WRITELN ('al trabajo a pie;¡sienta bien!')
 ELSE
   IF LLUVIA THEN
     WRITELN ('quedate en casa')
   ELSE
     WRITELN ('vete a jugar al tenis')
END.
```

La curiosa FUNCTION sí (sin parámetro esta vez; no hace falta) permite recoger la respuesta y transferir su resultado (lógico) a una expresión IF. Esto ocurre con la pregunta inicial, que se soluciona en seguida con beso o no beso. En cambio, en las preguntas siguientes se ha recurrido a variables booleanas temporales (trabajo y lluvia) para permitir, más adelante, el barrido casuístico.

Este programa nos ofrece la oportunidad de realizar algunas aclaraciones útiles sobre la estructura IF THEN/ELSE cuando aparece anidada. Dado que la primera regla es el sentido común, hay que tener en cuenta que:

- tanto IF como ELSE hacen de separadores (como BEGIN, END o mejor, como REPEAT, UNTIL);
- en los casos de instrucciones múltiples, las inherentes a una misma IF estarían encerradas entre dos separadores; si no habría problemas (errores lógicos o malentendidos con el compilador).

Un ejemplo clásico de incomprensión entre la intención del programador y la interpretación que de él realiza el compilador se presenta con la llamada IF "abreviada", o sea, sin ELSE.

Supongamos que en nuestro programa, por algún extraño motivo, se quiere eliminar la respuesta 'al trabajo a pie' (porque la oficina está demasiado lejos, por ejemplo). El principiante pensaría que es muy fácil y lo modificaría así:

```
IF trabajo THEN
  IF lluvia THEN
    WRITELN ('al trabajo en coche')
ELSE
  IF LLUVIA THEN
    WRITELN ('quedate en casa')
  ELSE
    WRITELN ('vete a jugar al tenis')
END.
```

Ahora bien, indicando las dos respuestas posibles con S y N, ¿qué salida dará ahora el programa en los cuatro casos posibles?

Intenten adivinarlo y compárenlo con lo que sigue:

SS——— 'al trabajo en coche'
SN——— 'vete a jugar al tenis'
NS y NN——— NINGUN MENSAJE!!!

¿Sorprendidos? En el segundo caso el mensaje es inmoral (ir al tenis en día laborable porque hay sol), mientras que en los dos últimos no hay respuesta. El hecho es que el compilador no hace el más mínimo caso de las indentaciones que utilicemos y asocia cada ELSE a la última IF que encuentra. ¿Y si se trata de IF sin ELSE? No se escandaliza y ejecuta solamente la primera instrucción que encuentra después de la cadena IF; se trata precisamente de la IF "abreviada" (utilizada ya en el caso del beso a la mujer). Veamos, para satisfacer a todos, los diferentes casos detalladamente:

- SS. Funcionan las dos primeras IF, mientras que la primera ELSE provoca el salto hasta END;
- SN. La primera ELSE se toma, también aquí, como alternativa a la lluvia, después de lo cual la nueva IF lluvia da vida a la otra ELSE, de aquí la respuesta;
- NS y NN. No hay ningún mensaje, porque no verificándose trabajo, se salta directamente a END. Si reflexiona descubrirá fácilmente que todo depende de que el compilador asocia, en todos los casos, la primera ELSE a la segunda IF.

¿Cómo podríamos remediarlo? Hay dos posibles soluciones para problemas de este tipo. La primera consiste en recurrir a expresiones booleanas:

```
IF trabajo AND lluvia THEN
    WRITELN ('al trabajo en coche')
IF NOT trabajo AND lluvia THEN
    WRITELN ('quedate en casa')
IF NOT trabajo AND NOT lluvia THEN
    WRITELN ('vete a jugar al tenis')
END.
```

La segunda solución consiste en encerrar entre un BEGIN y un END la segunda IF:

```
IF trabajo THEN
 BEGIN
  IF lluvia THEN
    WRITELN ('al trabajo en coche')
 END;
ELSE
  IF LLUVIA THEN
    WRITELN ('quedate en casa')
```

```
        ELSE
          WRITELN ('vete a jugar al tenis')
      END.
```

De esta manera se "convence" al compilador para que asocie la primera ELSE a la primera IF.

Finalmente, supongamos que a alguien se le ocurra arreglarlo poniendo un ";" después de WRITELN ('al trabajo en coche')... esto es lo peor que se podría hacer: el compilador se enfadaría porque no le saldrían las cuentas de las IF y de las ELSE.

Según dicen los críticos del Pascal, estos contratiempos se derivan de la ausencia de terminadores ENDIF. Existe una solución posible, aunque no muy limpia, que consiste en cerrar cada IF tipo abreviada con un ELSE (en efecto, si ELSE no está seguida de ninguna instrucción, el compilador no se escandaliza). En nuestro caso, el truco consiste en quitar el BEGIN y sustituir su END por ELSE.

Finalmente, a propósito de BEGIN y END, hay que recordar que si deseamos ejecutar varias instrucciones a continuación de un IF o de un ELSE, no se podrán olvidar, en absoluto. En efecto, aquí no tenemos lo que normalmente ocurre en BASIC, donde si en una línea se tiene

```
100 IF <CONDIC> THEN instr1:instr2:instr3:...
```

todas las instrucciones de la línea se ejecutan o se saltan.

El Pascal es más riguroso, y si nos olvidamos BEGIN y END asume que sólo debe ejecutar la primera instrucción.

La estructura CASE

El Pascal proporciona la estructura CASE para simplificar y aclarar las situaciones de elecciones múltiples. Es una construcción comodísima, siempre que a cada elección corresponda una sola instrucción. De todas formas, también se puede aplicar en otras situaciones, bien definiendo aparte procedimientos adecuados, o bien recurriendo al habitual dúo BEGIN-END.

El ejemplo que veremos con la tortuga sirve para moverla en la pantalla pulsando algunas teclas. Las posibilidades previstas son: movimientos hacia delante o hacia atrás de 2 pasos de longitud y rotaciones absolutas según la rosa de los vientos, o sea, 8 ángulos que van de 45 en 45 grados. Es evidente que si aquí no hubiera habido estructura CASE habría sido necesario inventarla o bien recurrir a una tabla, pero ésta sería una solución de

menor claridad semántica. Para las dos primeras acciones adoptamos las teclas A y S, y para las otras, las teclas que generalmente están dispuestas a lo largo de las 8 direcciones indicadas. Tendremos:

```
PROGRAM tortpaseo;
VAR caract:CHAR;
BEGIN (* main *)
 PENCOLOR(WHITE);
 READ(caract);
 WHILE caract<>CHR(13) DO
 (* si no es return *)
   BEGIN
    CASE caract OF
     'A':MOVE(2);
     'S':MOVE(-2);
     'J':TURNTO(0);
     'M':TURNTO(-45);
     'N':TURNTO(-90);
     'B':TURNTO(-135);
     'H':TURNTO(180);
     'Y':TURNTO(135);
     'U':TURNTO(90);
     'I':TURNTO(45);
    END;(* final CASE *)
    READ(caract);
   END;(* final de WHILE *)
END.
```

CASE es un ejemplo tan elocuente que no merece ningún comentario. Confirmar sólo que la construcción es del tipo:

```
CASE <expresion> OF <lista de casos> END
```

donde la lista de casos corresponde a los posibles resultados de la expresión. Es interesante la posibilidad de reagrupar juntos, separados por comas, casos que prevén una solución idéntica. En el programa precedente, por ejemplo, se podrían equiparar mayúsculas y minúsculas:

```
CASE caract OF
 'A','a':MOVE(2);
```

En el Pascal estándar falta la opción ELSE, asociada directamente a la CASE, que permita incluir, en una única acción, los casos no incluidos en la lista, incluso anidándolo todo. No se nota mucho su ausencia, pues generalmente se suele sustituir por una IF/ELSE por encima de la CASE.

Para quien nos siga con la debida atención, hacemos la siguiente pregunta: ¿qué ocurre si, en el programa precedente, se pulsa una letra que no está comprendida en la lista CASE?

Dejamos en el aire la facilísma respuesta, que confirmaremos al acabar el capítulo.

El problemático GOTO

Terminamos la relación de las estructuras de control con el discutido GOTO, eliminado ignominiosamente por los estructuralistas a ultranza, a pesar de que realmente el GOTO no es una estructura, sino un constructor de estructuras. Wirth fue benevolente, y sólo relegó el GOTO a ciudadano de segunda, de forma que todos los manuales de Pascal, siguiendo su ejemplo, lo tratan al final y "con prisas".

El GOTO se utiliza en casos extremos y complicados; por ejemplo, donde las estructuras disponibles vuelven excesivas la duplicación de códigos o el abuso de booleanas. Para hacer desistir del uso del GOTO, Wirth pensó en introducir el empleo, un poco molesto, de etiquetas (label) de destino; se componen sólo de cifras, pero son tipos y NO manipulables de ninguna forma y tienen que ser declaradas las primeras en el módulo al que pertenezcan. Lo explicamos con un ejemplo:

```
PROGRAM p;
LABEL 1;
....
  PROCEDURE A;
  LABEL 2;
  .....
  BEGIN
  .....
  GOTO 2
  .....
  GOTO 1
  .....
  2:WRITELN('pequeño error....')
  .....
  END;(* procedure a *)
  ......
  1:WRITELN ('¡error tragico!')
  ......
  END.
```


GOTO, obviamente, es asociable a IF, obteniéndose la familiar combinación: IF <condic> THEN GOTO...

En el pequeño ejemplo anterior, GOTO es útil para la salida anticipada de un bucle, especialmente (pero no sólo) cuando haya un error. Por lo tanto, aquí también es válido el principio de que el salto de un bloque interno a otro externo es lícito, pero no lo contrario.

De todas formas, una vez que se plantea un programa con construcciones WHILE, REPEAT, etc., que originan módulos quizá complejos pero bien engarzados unos en otros y con un único punto de entrada y salida, GOTO, además de superfluo, puede resultar complicado para su manejo. Si a esto añadimos efectos colaterales, sobre los que no podemos detenernos, pero que aconsejan mucha precaución, es posible que tengamos que decirle adiós al GOTO.

Acabamos este largo capítulo con dos asuntos. Ante todo, la respuesta a la pregunta sobre CASE: evidentemente, si se pulsa una tecla no prevista (y distinta del retorno del carro) la tortuga no se mueve.

El segundo asunto es la presentación de las cartas sintácticas de las figuras 5 y 6. La figura 5 contiene todas las construcciones del Pascal, mientras que la 6 resume la estructura de un "bloque" de programa. Recordamos nuevamente que la sucesión debe de respetar el orden Etiqueta, Construcción, Type, Var, Procedimiento y Function, que como truco nemotécnico, equivaldría a: "Esta Casa Tiene Varias Puertas Falsas".

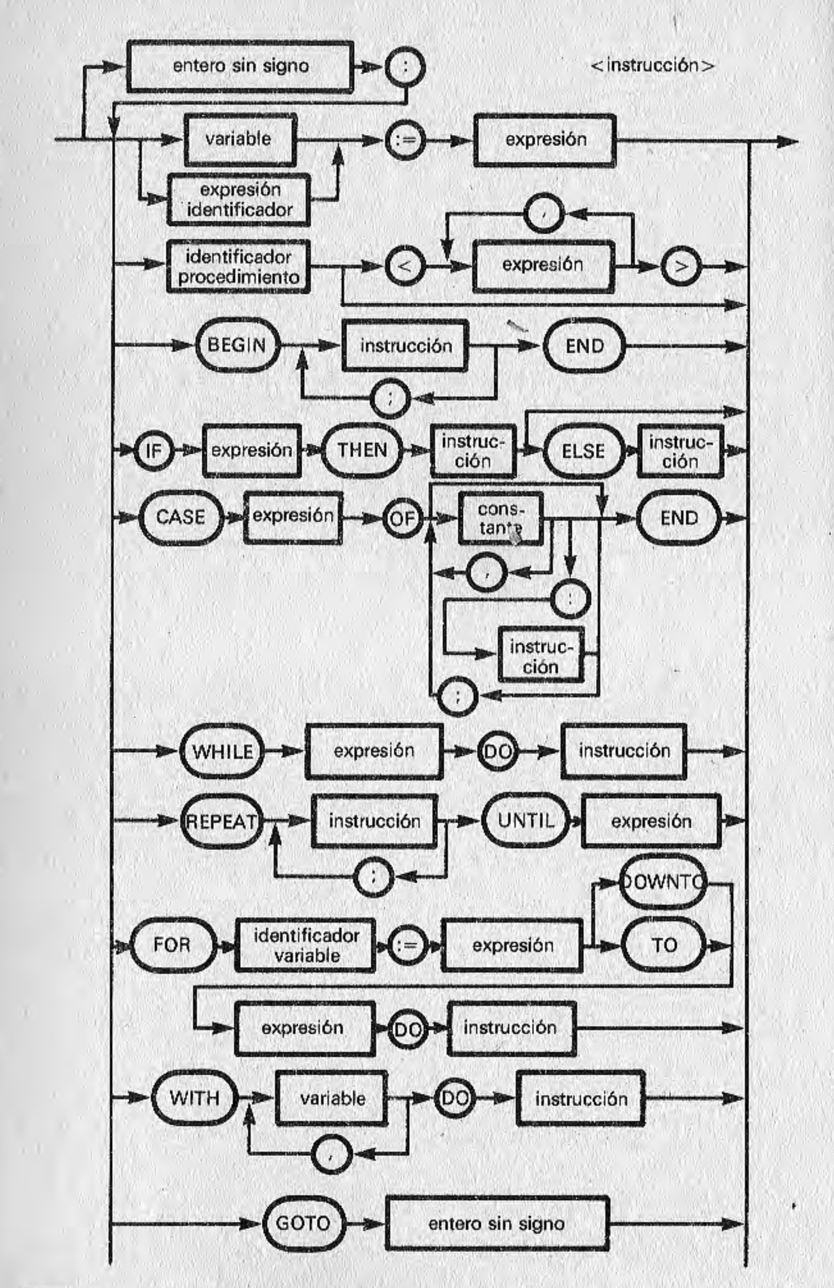


Figura 5.—Sintaxis de las estructuras de un programa Pascal. Para más detalles, le sugerimos consultar con manuales especializados.

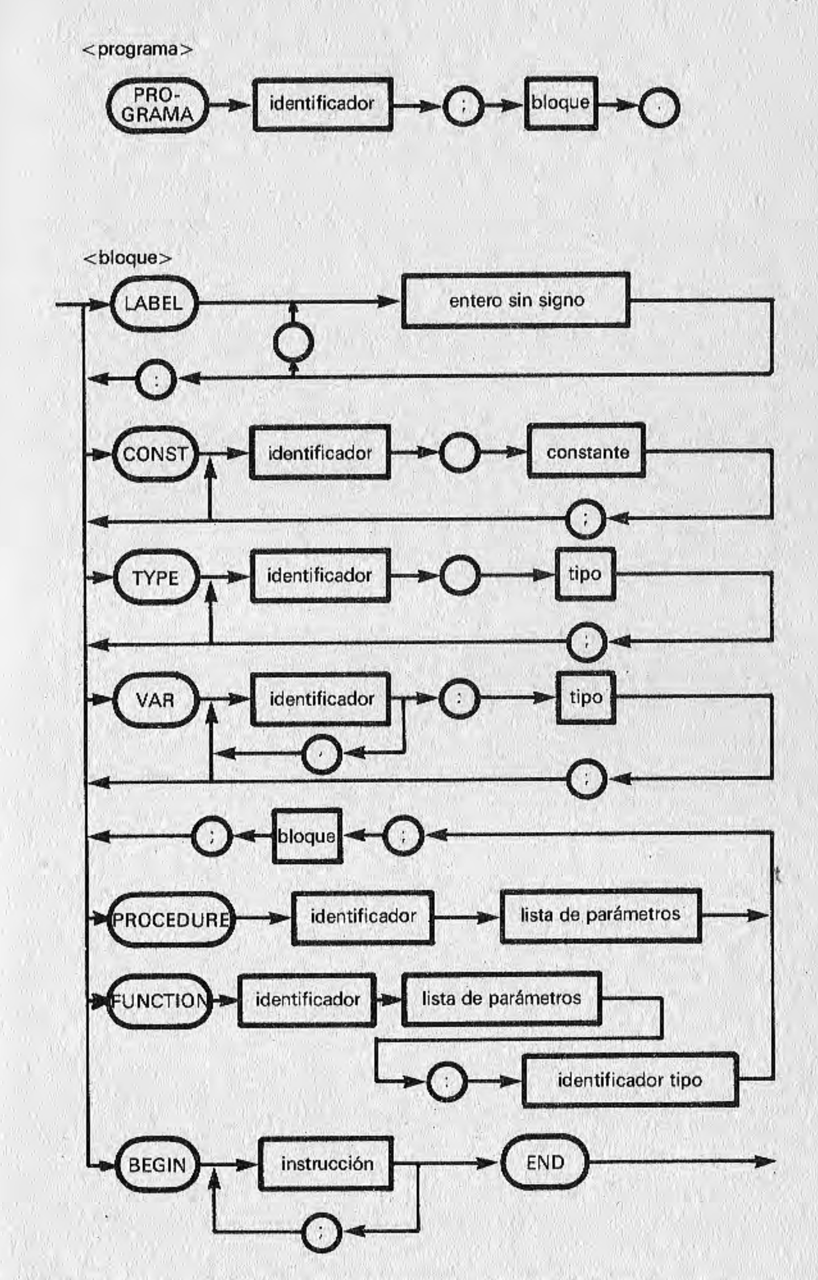


Figura 6.—Sintaxis de la estructura de un bloque de programa en Pascal.

CAPITULO V

FUNCIONES, PROCEDIMIENTOS Y LOS EXTRAÑOS ANILLOS DE LA RECURSIVIDAD

Seamos serios...



Antes de afrontar el complejo tema de los tipos de datos que, según lo que ya hemos visto en los dos capítulos introductorios están bastante ligados a los algoritmos , nos proponemos ampliar nuestros conocimentos sobre las FUNCTION y acercarnos al mundo de la recursividad. Veamos, pues, un par de ejemplos relativos a las FUNCTION. El primero tiene un cierto carácter práctico. Se refiere al cálculo mensual del IRPF correspondiente a la base imponible de un asalariado, es decir, a la parte del salario bruto que queda después de haber descontado todos los pagos a la Seguridad Social.

Nótese que el algoritmo se basa sobre unas tablas aproximadas, que no coincidirán seguramente con las que usted deberá emplear al liquidar su Declaración de la Renta de 1985. Paciencia, pues se trata simplemente de un ejercicio. El criterio de impuestos progresivos prevé distintos tramos impositivos:

Tramo impositivo	Tabla de retención
Hasta 25000	10 %
33333	13 %
41666	16 %
49999	19 %
62500	22 %
75000	25 %
....	....

La columna de la derecha expresa el porcentaje que se aplica a la base imponible comprendida entre el valor de la izquierda y el siguiente. El procedimiento que vamos a ver se basa en la observación de que ambas columnas tienen valores que se suceden con regularidad: los porcentajes de la tabla crecen un 3%, mientras que los tramos impositivos crecen a razón de 8.333 hasta 49.999, y desde aquí se incrementan en 12.500 pesetas (lo extraño de estos números se debe a que se trata de valores anuales referidos a meses).

El algoritmo que sugerimos aquí consiste en lo siguiente:

- aplíque un 10% sobre la base imponible;
- considére un posterior 3% sobre la diferencia entre ésta y cada tramo impositivo, hasta que lleguemos al que corresponde a la base imponible.

Realizadas estas hipótesis, escribir el programa es cosa fácil. Como de costumbre, animamos a los más voluntariosos a intentarlo. Sólo una obligación: utilizar una FUNCTION de nombre Impbrut. He aquí la solución propuesta:

```
PROGRAM IMPUESTOS;
VAR SUELDO BRUT,RETENC,IMPONIB,IMP,IMPENET:INTEGER;
  FUNCTION IMPBRUT(IMPONIB:INTEGER):INTEGER;
  CONST INICTABLA=25000;INTERMED=49999;
        CREC1=8333;CREC2=12500;
  VAR TRAMIMP,IMP:INTEGER
 BEGIN
  IMP:=0;
  TRAMIMP:=INICTABLA;IMPBRUT:=IMPONIB DIV 10;
  WHILE IMPONIB>TRAMIMP DO
   BEGIN
    IMPBRUT :=IMPBRUT+TRUNC((IMPONIB-TRAMIMP)*0.03);
    IF TRAMIMP<INTERMED
      THEN TRAMIMP:=TRAMIMP+CREC1
      ELSE TRAMIMP:=TRAMIMP+CREC2
   END (* WHILE *) IMPBRUT:=IMP
  END; (* IMPBRUT *)
 BEGIN (* MAIN *)
  ....
  READLN (SUELDOBRUT,....);
  ....
```



```
  IMP:=SUELDOBRUT-RETENC;
  IMP:=IMPBRUT(IMP);
  ....
  END.
```

Obviamente, los "puntitos" corresponden a "omisiones" que cada uno puede rellenar para su uso particular.

En este ejemplo, al haber sido más concretos, se hacen evidentes, precisamente por esto, ciertas ventajas aparentemente formales del Pascal. Es útil, aunque no indispensable, definir variables locales de FUNCTION en cuanto sirven exclusivamente para el uso interno del cálculo fiscal. De esta forma se evita cualquier riesgo de que en el main se modifiquen variables que NO deben de ser influidas. Este aspecto es decisivo en la utilización recursiva de procedimientos y funciones que vamos a ver ahora, y en el caso de módulos de "librería" reutilizables, como en el lenguaje Modula 2. También las constantes inictabla, intermed, crecl y crec2 permiten una cómoda actualización en el caso de que deban modificarse posteriormente (por ejemplo, para reutilizar la FUNCTION en el cálculo del IRPF anual en vez de mensual); es suficiente cambiar sólo la sección CONST del módulo en lugar de todas las fórmulas afectadas por el cambio (que pueden ser muchas en el caso de programas importantes). Se confirma así la mayor transparencia de un listado Pascal con respecto a uno en BASIC. Aunque no tenemos reparos en admitir nuestra afición por el popular BASIC y en apreciar sus dotes de inmediatez y sencillez, sin embargo tenemos que proclamar que el Pascal, aunque a menudo necesita mayor cuidado y tiempo para ser definido y puesto a punto, ofrece al final unos resultados autodocumentados, haciendo innecesarios gran parte de los comentarios o diagramas de flujo explicativos.

Habrá notado en el ejemplo la adopción de variables tipo INTEGER. Esto se debe a que en España la moneda no emplea decimales y, además, a que los números de 6-7 cifras que hemos utilizado, expresados en tipo REAL, darían lugar a resultados en forma exponencial, del tipo $1.0000E^6$ (en realidad, el tipo INTEGER tiene un rango en el UCSD que se limita, aproximadamente, a 32.000, valor a partir del cual sería necesario adoptar el tipo LONG). En este momento se tropieza con la rigidez del Pascal en materia de tipos: generalmente está prohibido mezclar "manzanas con peras"; sólo se permite juntar reales y enteros, con prioridad siempre para los primeros. Además, si el primer miembro es de tipo entero y el segundo tiene algún término real se produce un error, es decir, no se realiza la transformación automática a tipo INTEGER. Por suerte, el Pascal proporciona la función TRUNC (que ya hemos uti-

lizado en la fórmula para obtener el total de impbrut). Además de la DIV, ya vista, podemos usar la operación MOD, que proporciona el resto de la división entre enteros (ej.: resto := dividendo MOD divisor).

Queremos aprovechar para aclarar que entre los objetivos de este libro introductorio no se encuentran temas como:

- cálculo numérico,
- elaboración de las E/S en general;
- o tratamiento de archivos.

Para estos puntos, o para ampliar detalles sobre los vistos, le recomendamos la lectura de manuales más especializados.

Volviendo al ejemplo, hacemos notar que también se podrían haber insertado instrucciones como:

```
IMPNET:=IMPBRUT(IMP)-DESGR;
WRITELN ('EL IMPUESTO NETO RESULTA IGUAL A: ',IMPBRUT(IMP)-DESGR);
```

Recurrir a la recursividad

Todo lo dicho puede ayudar a reconsiderar la diferencia entre FUNCTION y PROCEDURE. Dejamos como tarea para los más voluntariosos la implementación del mismo programa precedente por medio de una PROCEDURE que, naturalmente, contenga un parámetro-variable oportuno, algo así como:

```
PROCEDURE IRPF(VAR IMPBRUT: INTEGER,IMPONIB:INTEGER);
```

Nos queda por comentar una última diferencia entre los dos módulos pascalianos fundamentales: el tratamiento de la recursividad. Dado que es uno de los argumentos más fascinantes, y en cierto modo misteriosos, de lenguajes evolucionados como el Pascal, merece la pena que le hagamos un hueco en este libro. En la eterna disputa entre BASICalistas y Pascalistas, los segundos, a las denigraciones de los primeros ("¿Dónde están las cadenas?, ¿dónde los archivos aleatorios?, ¿dónde...?") reaccionan sistemáticamente encogiéndose de hombros y aludiendo con dignidad a este potente y extraño instrumento.

Para empezar a presentarlo hay que comentar su estrecha relación con los discursos sobre discursos. Ejemplo trágico: dijo Epaminonda el Cretense: "lo que estoy diciendo es falso". Ejemplo humorístico: "Había una vez un Rey que dijo a su criada: «¡Cuéntame un cuento!», y ella empezó: «Había una vez un Rey que dijo a su

criada: «¡Cuéntame un cuento!», y ella empezó: «Había una vez un Rey...»"

La serie de comillas que cierran el segundo caso hay que suponerla infinita, en tanto que este cuento, formado por una cadena de reyes que mandan hablar a criadas que hablan de reyes que mandan hablar a criadas que hablan de..., evidentemente no acaba NUNCA. En cuanto al primer caso, los lectores más estudiosos, esos que, quizá esporádicamente, lean "Investigación y Ciencia" (de "Scientific American") saben ya que da lugar a la famosa contradicción, en base a la cual el ambiguo Epaminonda si dice lo verdadero dice también lo falso, y viceversa. Pero dejemos de divagar y volvamos al Pascal.

Dicho con palabras, la recursividad, en informática, consiste en que un procedimiento dado puede, desde su interior, llamarse a sí mismo. El juego, como se intuye, podría durar hasta el infinito, pero se insertan criterios adecuados siempre internos de finalización o bien una condición de final que se integra más o menos implícitamente en la definición.

En vista de que dicen que la tortuga se ha creado precisamente para demostrar lo fácil que es comprender, incluso para los niños, asuntos como la recursividad, empezaremos con un ejemplo geométrico clásico con la tortuga:

```
PROGRAM CUADRADOS;
 PROCEDURE CUADRO(LADO,INC,ALFA:INTEGER);
 VARI:INTEGER;
 IF LADO<=200 DO
 BEGIN
   TURN(ALFA);
   FOR I=1 TO 4 DO
   BEGIN
    MOVE(LADO);TURN(90)
   END;
   CUADRO(LADO+INC,INC,ALFA)
 END;
BEGIN (* MAIN *)
 MOVETO(120,100);PENCOLOR(WHITE);
 CUADRO(5,2,6)
END.
```

Después de lo dicho, el resultado del programa no será ni imprevisible ni chocante: partiendo del centro de la pantalla, y debido a la primera llamada a Cuadro por parte del main, se traza un cuadrado de 5 unidades de lado y con una desviación de

16 grados, terminado lo cual la cosa se repite porque el Cuadro se evoca a sí mismo, pero esta vez con un parámetro "lado" incrementado en 2. Según esto, ¿hasta dónde podríamos llegar? Teóricamente hasta el infinito, con cuadrados cada vez mayores, si no fuese por el providencial IF, que bloquea el proceso cuando el lado es mayor que 200.

Cambiando los parámetros podemos ver los distintos efectos que provocan siguiendo la peligrosa costumbre de quienes se ponen a jugar con la tortuga. Por ejemplo, sugerimos desplazar la instrucción Cuadro (lado + inc, inc, alfa) autoinvocadora antes del END que la precede, de forma que ahora la autollamada ocurra después de cada MOVE (lado) y TURN (90):

```
FOR I=1 TO 4 DO
BEGIN
 MOVE(LADO);TURN(90)
 CUADRO(LADO+INC,INC,ALFA)
END (* BUCLE FOR *)
END;
```

De este pequeño ejemplo podemos extraer dos conclusiones:

- en algunos casos la recursividad produce efectos que se obtienen más fácilmente con una simple iteración;
- por tanto, es mejor usar esta técnica sofisticada sólo en casos especiales.

En consecuencia, antes de continuar desarrollándola sin ton ni son, vamos a intentar profundizar en este argumento.

FUNCTIONS y PROCEDURES recursivas

Conviene hacer en seguida una pequeña prueba para satisfacer una duda que seguramente ha surgido en muchos de ustedes. Eliminen en el ejemplo precedente (en su forma inicial) el IF, sin olvidar los respectivos BEGIN... END y anulando incluso la expansión del cuadrado (haciendo en el main: Cuadro (20, 0, 6)). ¿De verdad durará hasta el infinito el trazado de estos 360/6 = 60 cuadrados? La respuesta es negativa: después de un cierto tiempo el sistema producirá un mensaje de "stack overflow" (superación de la capacidad de la pila) y se parará. Efectivamente, esto puede ayudar a entender mejor lo que estamos a punto de decir: el mecanismo de la recursividad acumula en una pila ("stack") los parámetros y variables locales pendientes de resolución. Ahora bien,

toda pila dispone de un área finita de la memoria y (a menos que se pongan en juego mecanismos aún más sofisticados...) ésta, antes o después, se saturará, llegando a desbordarse ("overflow").

Veamos un caso sencillísimo: el de un sumatorio, desde "1" hasta "n", a cuyo total llamaremos Tot(n). La definición, también en matemáticas, se puede dar así:

```
TOT(N)=1 SI N=1 ,EN CASO CONTRARIO
TOT(N)=TOT(N-1)+N
```

Así, por ejemplo, si n = 6 tendremos Tot(6) = Tot(5) + 6. ¿No es banal? Ciertamente, pero la sutileza está en reconocer que esta definición ES RECURSIVA POR SI MISMA, en cuanto que nos obliga implícitamente a volver a un cálculo, aunque sea en potencia y no de facto. Generalmente se prefiere la forma directa, calculando primero Tot(1), después Tot(2) = Tot (1) + 2, etc. La esencia del programa es obvia:

```
....
TOT:=1;
FORI:=1 TO N DO
TOT:=TOT+I
```

Pero, ¿por qué no obligar al ordenador a hacer cuentas definidas recursivamente? En Pascal lo hacemos en seguida:

```
PROGRAM SUMAREC;
VAR N,I,TOTAL:INTEGER;
 FUNCTION TOT(NUMERO:INTEGER):INTEGER;
 BEGIN
  IF NUMERO=1 THEN TOT:=1
  ELSE TOT:=TOT(NUMERO-1)+NUMERO;
 END;
BEGIN (* MAIN *)
REPEAT
 READLN(N);I:=N;
IF N<=4000 THEN
 BEGIN
   TOTAL:=TOT(N);
   WRITELN;(* RETORNO DE CARRO *)
   WRITELN ('SUMATORIO HASTA ',N,' =',TOTAL);
 END
UNTIL N>4000
END.
```

La clave de este asunto reside en el hecho de que cuando en el segundo miembro de una expresión se encuentra un término que incluye el nombre de una FUNCTION pascaliana se realiza una llamada a dicha función. Esto ya lo habíamos visto a propósito de un WRITELN (y lo volveremos a ver); ahora nos lo encontramos en una expresión interna a la FUNCTION misma, con lo cual supone una recursividad. El juego recursivo consiste en que la autoevocación se realiza cada vez con valor distinto (una unidad menor) del parámetro "número". La condición de final del proceso se cumple cuando el valor "número 1" de llamada es 1. Quien "se fíe" del hecho de que el Pascal sostiene la recursividad tiene que "creer" que estas cosas suceden, y por otro lado, la prueba empírica se puede obtener simplemente haciendo correr el programa y proporcionándole diferentes valores de "n". A propósito: habrá notado una variante del usual WHILE mediante REPEAT que, por esta vez, permite utilizar un solo READLN. De todas formas, mediante tanteo, podrá lograr la parada del programa por "stack overflow" (superación de la capacidad de la pila) para valores altos de "n" (alrededor de 2000 para un ordenador personal corriente). Esto estimula de nuevo nuestra curiosidad sobre el tema, además de recordarnos que no debemos abusar de la recursividad, especialmente en los pequeños ordenadores domésticos. Para intentar comprender por nosotros mismos este mecanismo, les invito a incluir una nueva línea después del ELSE de Tot:

```
I:=I-1;WRITELN('RASTRO: ',I);
```

¿Por qué se ha adoptado una variable global? Pues porque no era muy sencillo hacerlo de otra forma: una variable local hubiera sido más adecuada, pues se quiere seguir las huellas de lo que sucede en el sitio, pero se presentaba muy enmarañado el problema de la inicialización, por lo menos para un principiante. Si ponemos en el main el valor inicial "i" igual a "n", la instrucción arriba vista parece la más lógica. En la práctica, ¿qué es lo que sucede? He aquí la salida del programa en el caso n = 4:

```
Rastro: 3
Rastro: 2
Rastro: 1
Rastro: 0
Sumatorio hasta 4 = 10
```

Seguramente el 0 haya producido alguna sorpresa. Veamos qué ocurre si modificamos la nueva línea de esta manera:

```
I:=I-1;WRITELN('RASTRO: ',NUMERO,' ',I);
```


El resultado de la ejecución sería:

```
Rastro: 1 3
Rastro: 2 2
Rastro: 3 1
Rastro: 4 0
Sumatoria hasta 4 = 10
```

Esperamos no haber causado ningún ataque cardíaco. Aunque en principio resulte increíble, la asociación de valores para "número" e "i" es correcta. En efecto, la recursividad deja pendiente la realización de todos los cálculos que *no pueden* ser ejecutados (cada vez que el parámetro "número" decrece), pero conserva los valores con los que habrá de hacerlos en la pila. Así, cuando llegamos a la presencia de la condición IF, con número = 1 tendríamos problemas si no se hubiesen conservado uno a uno todos los valores precedentes de "número" con los cuales hay que trabajar ahora. Cuando son precisos se repescan *en orden inverso* al que han sido "apilados" en la pila usada por el Pascal. En el caso de n = 4 poco a poco tendremos que añadir a Tot:=1, 2, luego3 y 4, en este orden.

Los que aún no están del todo convencidos preguntarán: ¿por qué entonces los valores de "i" van desde 3 a 0? Esto depende del hecho de que "i" es una variable global y, por lo tanto, NO está almacenada en la pila. Sin embargo, la ejecución de la instrucción i:=i-1 y los sucesivos WRITELN han quedado en reserva por las llamadas "de telescopio" del Tot (número-1) y sólo cuandoo número=1 empezará a decrecer i: ésta es la explicación de porqué "i" va desde 3 a 0, decrementándose 4 veces antes de volver definitivamente al main.

En la figura 1 se representan, en cuatro planos distintos, las sucesivas variaciones de la pila, con la operación evocada (y en reserva) y los parámetros apilados. Desde un punto de vista general, todo lo que es "local" —es decir, variables y parámetros— se reserva en la pila; es por esto por lo que, en problemas en los que estan en juego un mayor número de parámetros, la pila se "derrumba" antes.

Las flechas que apuntan hacia arriba, a la izquierda, simbolizan las 4 operaciones sucesivas de PUSH (empujar literalmente), mientras que a la derecha se representan palabras condicionales como IF o ELSE. La presencia de IF en la cima (último plano) da lugar a un desbloqueo de la situación (que en otro caso llegaría al infinito o, en realidad, al desbordamiento de la pila). Entonces se activa el mecanismo de cálculo hacia atrás, con los 4 POP (ex-

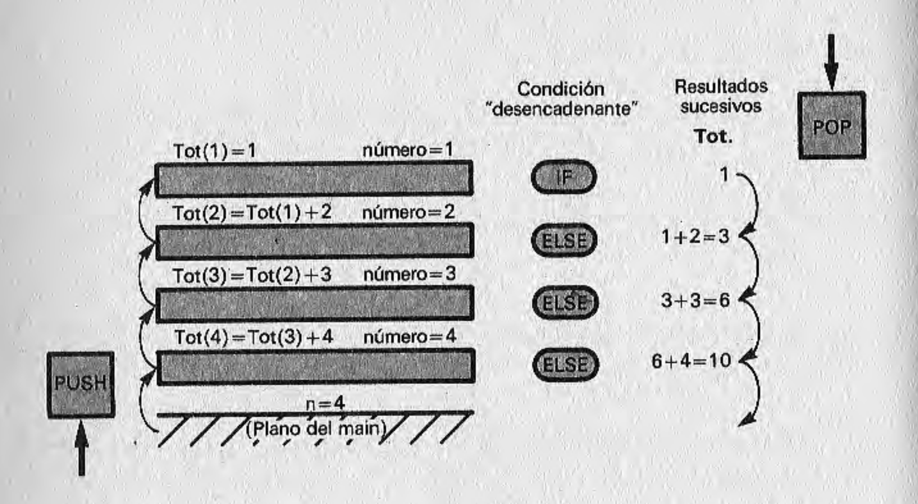


Figura 1.—La figura trata de evidenciar el funcionamiento de la pila interna (stack) guardando parámetros y variables locales y manteniendo en reserva fórmulas cuando se invoca una recursiva, como en el programa del sumatorio.

traer, lo contrario de Push), necesarios para volver al main, con la consiguiente recuperación de los distintos valores.

Por último, veamos otra posible variante, adoptando un PROCEDIMIENTO (naturalmente con un parámetro-variable). Aunque es evidente que para un caso analítico es preferible una FUNCTION, proponemos esto como un útil ejercicio explicatorio. Nuestra solución es ésta:

```
PROGRAM SUMAREC;
VAR N,SUMA:INTEGER;
PROCEDURE TOTAL(VAR TOT:INTEGER;NUMERO:INTEGER);
BEGIN
 IF NUMERO=1 THEN TOT:=1
  ELSE
   BEGIN
    TOTAL(TOT,NUMERO-1);TOT:=TOT+NUMERO;
   END;
  WRITELN ('RASTRO: ',TOT,' ',NUMERO)
END;
BEGIN (* MAIN *)
.....
```



```
IF N<2000 THEN
BEGIN
 TOTAL(SUMA,N);WRITELN (SUMA)
END
UNTIL...
END.
```

Ahora los comentarios casi sobran. La distinta estructura nos ha hecho introducir una nueva variable global 'suma' para hacerla pasar como parámetro "de referencia" al procedimiento Total. Así podemos subrayar por fin la última diferencia entre FUNCTION y PROCEDURE: en las operaciones de cálculo, la primera está, por así decirlo, mejor dispuesta para la recursividad, mientras que la segunda debe llamarse explícitamente y debe ser seguida por la instrucción que maneja el parámetro-VAR; el mecanismo también está garantizado, pero al estar la segunda operación en reserva por la evocación de Total, automáticamente queda guardada en la pila habitual (una pregunta: ¿qué habría sucedido si nos hubiéramos olvidado de encerrar las dos operaciones entre BEGIN y END...?

Quizá exista la pequeña ventaja de la mayor comodidad con la que se puede tener ahora un rastro de los cálculos "hacia atrás" (aquí conviene hacer otra pregunta: ¿cuál es el mejor lugar para situar la instrucción siguiente:

WRITELN ('Rastro: 'tot, ''número);?Hay que razonarla antes de probarla en el ordenador personal.

Arboles y dragones

Esta larga y prolija charla sobre temas elementales tenía fines únicamente pedagógicos, pues, como muchos de ustedes saben, el sumatorio de los enteros de "1" a "n" viene dado directamente por la fórmula.

S(n) = n*(n + 1)/2
Ejemplo: S(4) = 4*5/2 = 10

Como ya hemos aprendido muchas útiles sutilezas podemos pasar a desarrollar bastante velozmente ejemplos más complejos. Aunque nos limitaremos a casos gráficos de la tortuga, pueden estar seguros de que la recursividad encuentra aplicaciones muy serias, sean o no numéricas, en los más diversos campos, teniendo como terreno privilegiado el de la llamada Inteligencia Artificial,

permitiendo mecanismos de búsqueda en árboles (decisiones, movimientos estratégicos y similares) que, en caso contrario, serían extremadamente incómodos, por no decir imposibles de realizar en forma iterativa. Generalmente, y a grandes rasgos, se puede afirmar que la recursividad es cara en términos de memoria (por la gran cantidad de memoria que necesita para la pila) y de velocidad de ejecución (por culpa de los PUSH y POP, aunque esto puede variar de un problema a otro), pero tiene a su favor la ventaja de su potencia expresiva, además de computacional, y su carácter sintético.

Pasemos ahora al examen de otro caso interesante, muy citado también en los manuales de Logo.

```
PROGRAM CURVASDELDRAGON;
USES TURTTLEGRAPHICS;
VAR NIV:INTEGER;
 PROCEDURE DRAGON(NIVEL,LADO:INTEGER);
  BEGIN
   IF NIVEL=NIV THEN
   MOVE(LADO)
   ELSE
    BEGIN
     TURN(45);
     DRAGON(NIVEL+1,ROUND(0.707*LADO));
     TURN(-90)
     DRAGON(NIVEL+1,ROUND(0.707*LADO));
     TURN(45)
     END;(* ELSE *)
   END;
BEGIN (* MAIN *)
  INITTURTLE
  FORNIV:=0 TO 8 DO
  BEGIN
   PENCOLOR(NONE);MOVETO(70,50);
   PENCOLOR(WHITE);DRAGON(1,140);
   READLN
  END (* FOR *)
 END.
```

Se han puesto a propósito las instrucciones para hacer funcionar el programa concretamente con el Pascal del Apple, con el fin de permitir al menos a los poseedores de este ordenador realizar un experimento, lo que siempre ayuda a comprender el

mecanismo. En efecto, las 8 veces que se realiza el ciclo FOR del main se trazan otros tantos fragmentos recursivos de la curva, interrumpidos por la instrucción READLN, así que apretando Return se les puede ver perseguirse uno a uno. Si, en cambio, quiere ver dibujado todo a la vez, basta con suprimir READLN.

Examinando ahora la definición del procedimiento Dragón se ve inmediatamente que se compone de tres fases: un giro de 45 grados más la ejecución de Dragón, un giro de 90 grados en el sentido de las agujas del reloj, y luego, de nuevo, el Dragón seguido de una rotación de 45.

En ambos casos hay auto-evocación, con reducción del parámetro lado (pasado inicialmente con un valor de 140 desde el main) mientras que el parámetro nivel es incrementado en una unidad.

En base a lo que hemos dicho hasta ahora no es demasiado difícil darse cuenta de lo que ocurre después de los dos primeros pasos del bucle FOR, o sea, cuando la variable global "niv" tiene valores 0 y 1. En el primero se verifica la condición nivel=niv, luego ELSE ni siquiera se toma en cuenta y se traza un segmento horizontal de lado 140, devolviendo el control al main. Cuando niv, en el segundo paso, vale 1, se verifica la condición que activa ELSE, puesto que ahora nivel=0 y niv=1. Por lo tanto, se conecta el mecanismo de recursividad con un "lado" 0.707 veces (redondeado en el interior con la función ROUND) el valor que le ha sido pasado por el main. Con la segunda auto-evocación se completa la triple fase descrita más arriba. Para todos aquellos que saben que 0.707 (el inverso de la raíz de 2) es el coeficiente que liga el lado con la diagonal del cuadrado no será difícil comprender que tendremos el trazado de la curva señalada con "nivel 1" en la figura 2. Dos observaciones:

- a cada nuevo paso del FOR la pareja de instrucciones PENCOLOR (NONE) y MOVETO (70,50) vuelve a poner a la tortuga en el mismo punto de salida;
- como es evidente en la figura 2 la tortuga acaba su viaje con un ángulo relativo final cero (+45 -90 +45).

Si llamamos "diente de dragón" a este triángulo rectángulo (la punta del diente forma un ángulo de 90), comprender los movimientos sucesivos significa darse cuenta de que las cosas van de tal foma que se traza un diente de dragón sobre cada uno de los lados que competen al nivel inmediatamente precedente.

Nos falta espacio para un análisis más detallado y por eso les remito al ejemplo anterior. Aquí nos limitaremos a hacer una observación importante, sin la cual cualquiera se puede despistar fácilmente. Las llamadas recursivas a Dragón (nivel+1, ROUND (0.707

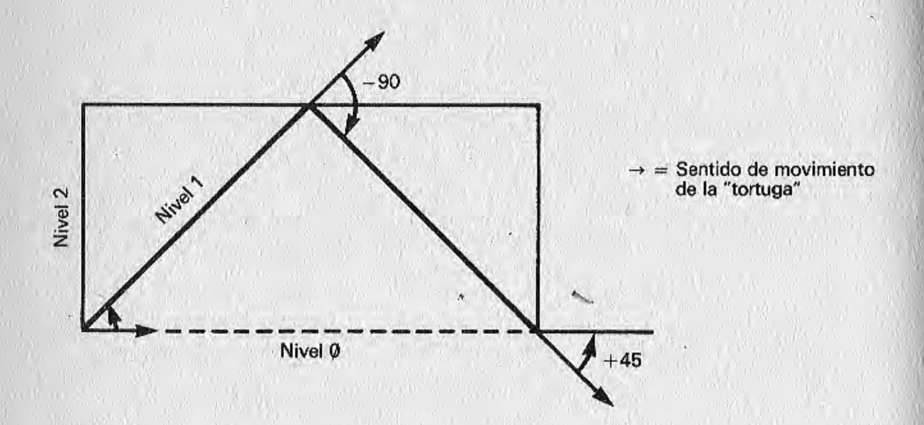


Figura 2.—Recorridos de la tortuga en el programa CurvasdelDragón en los valores 0, 1 y 2 de nivel. En el caso del nivel 1 se ponen en evidencia las rotaciones relativas de la tortuga. El ángulo final resultante es nulo.

* lado)) son dos; por lo tanto, nos preguntamos: cuando la primera permanece en reserva, ¿hay que seguir repitiéndola hasta llegar a nivel=niv o hay que pasar a examinar la segunda llamada a Dragón? La respuesta correcta es la primera, como puede comprobar si tiene la paciencia suficiente con la pila, aunque basta con "convencerse" de que el compilador ve como un todo aquello que hay entre el BEGIN y el END del ELSE, dejando, por así decirlo, en reserva el cuadro ejecutivo entero.

Concluyendo, en el nivel 2, por ejemplo, tendríamos la siguiente sucesión de hechos: giro de 45 (nivel 1 en reserva): giro 45 (niv 2), verificación de IF y consiguiente ejecución del primero de los 4 lados; giro de -90 (niv 2), segundo lado (debido al segundo Dragón que también está en reserva), y giro de 45 (con lo que la tortuga ahora está paralela al primer lado del niv 1); sigue un giro de 90 (¡ahora nivel 1!), etc.

Una última observación: las varias instrucciones TURN, que no son procedimientos recursivos, aunque estén en reserva (podríamos decir que por culpa de otros), al final son atendidas por igual, mientras que MOVE (lado), condicionada por IF, es ejecutada solamente en el último nivel. Si alguien no se ha convencido de esto, puede intentar hacer seguir READLN por una instrucción INITTURTLE (CLEARSCREEN en otros sistemas) o incluso puede probar a eliminar el FOR con una única instrucción de asignación tipo niv:=6, se dará cuenta de que sólo se dibuja una curva.

Otras indicaciones

La figura 3 ilustra el dibujo que se obtiene haciendo correr el programa precedente hasta el nivel 5. Por encima se obtienen efectos imprecisos debidos a la aproximación de la función ROUND (y, en definitiva, a la baja resolución gráfica). Se pueden obtener variantes de diferentes maneras, por ejemplo, cambiando el diente del dragón, introduciendo elementos pseudo-aleatorios y así sucesivamene. Otros criterios pueden ser del tipo ilustrado por el programa de la figura 4 y su correspondiente ejecución (Fig. 5).

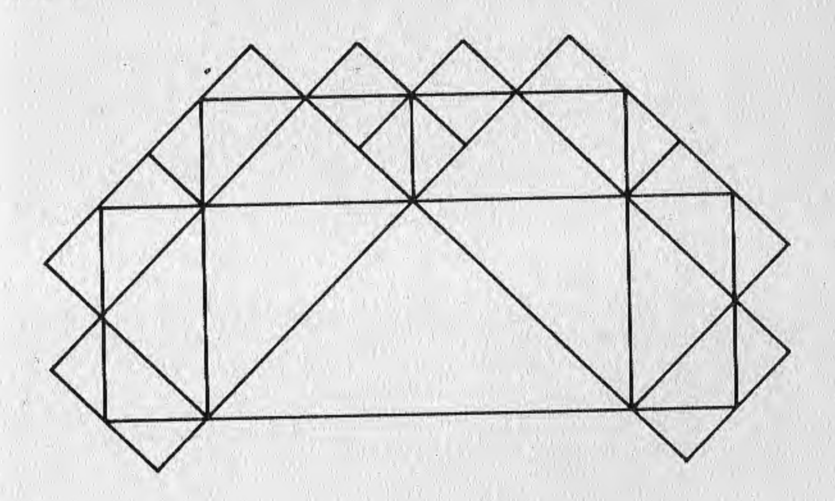


Figura 3.—Curvas del Dragón hasta el nivel 5.

También se habla de recursividad mutua entre procedimientos... Nos parece oír que alguien murmura una pregunta: ¿pero todas estas extrañas cosas, en la práctica, para qué sirven? Nuevamente repetimos que la recursividad sirve para muchos otros campos, como la búsqueda de datos en estructuras de árbol, también en la base de la Inteligencia Artificial, donde el barrido con técnicas recursivas de árboles de decisión (por ejemplo, el estudio de los movimientos en un juego) está a la orden del día, e incluso es el fundamento de lenguajes especiales como el LISP y el Prolog. En diferentes manuales aparece la solución recursiva del brillante e instructivo juego de las Torres de Hanoi, que, por lo tanto, nos limitamos a citar.

```
PROGRAM ARBORESCENCIA;
VAR ESCALA,ORDEN:INTEGER;

PROCEDURE ARBOL (X,Y,LONG,DIR:INTEGER);
PROCEDURE CAMBIAXY;
VAR DELTAX,DELTAY:INTEGER;
BEGIN
 IF DIR<0 THEN DIR:=DIR+8;
 IF DIR>=8 THEN DIR:=DIR-8;
 CASE DIR OF
   0: BEGIN DELTAX:=1;DELTAY:=0;END;
   1: BEGIN DELTAX:=1;DELTAY:=1;END;
   2: BEGIN DELTAX:=0;DELTAY:=1;END;
   3: BEGIN DELTAX:=-1;DELTAY:=1;END;
   4: BEGIN DELTAX:=-1;DELTAY:=0;END;
   5: BEGIN DELTAX:=-1;DELTAY:=-1;END;
   6: BEGIN DELTAX:=0;DELTAY:=-1;END;
   7: BEGIN DELTAX:=1;DELTAY:=-1;END;
 END; (* CASE *)
 X:=X+ESCALA*LONG*DELTAX;
 Y:=Y+ESCALA*LONG*DELTAY;
END; (* CAMBIAXY *)

BEGIN (* ARBOL *)
  PENCOLOR(NONE)
  MOVETO(X,Y);TURNTO(DIR*45);
  PENCOLOR(WHITE);
  CAMBIAXY;MOVETO(X,Y);
  IF LONG>1 THEN
   BEGIN
    ARBOL(X,Y,LONG-1,DIR+1);
    ARBOL(X,Y,LONG-1,DIR-1)
   END;
END;(* ARBOL *)

BEGIN (* MAIN *)
 WRITE ('ESCALA ? ');READLN (ESCALA);
 WRITE ('ORDEN ? ');READLN (ORDEN;
        ARBOL(0,-ESCALA*ORDEN-100,ORDEN,2);
END.
```

Figura 4.—El programa Arborescencia ilustra eficazmente el concepto de recursividad.

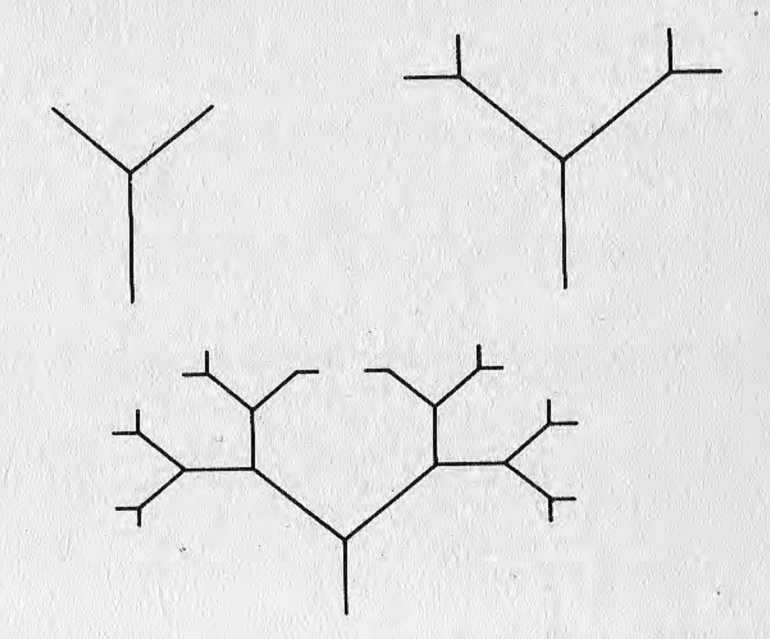


Figura 5.—Arboles de nivel 1, 2 y 4 creados por el programa Arborescencia, cuyo listado aparece en la figura 4.

El asunto se sale de los objetivos de este libro; basta con señalar qué técnicas gráfico-recursivas están en la base de los llamados "fractales", líneas recortadas generadas por el ordenador que, variando pacientemente diferentes parámetros y combinándolas sabiamente con una pizca de aleatoriedad, dan lugar a sorprendentes imágenes que evocan montañas, valles, paisajes lunares fantásticos e irreales. ¡El arte computerizado es una mezcla fascinante de mecánica y creatividad!

¿Y el BASIC?

Nos podríamos preguntar si es del todo exacto que el BASIC no soporta la recursividad. Probemos a verificar el funcionamiento de este programa:

```
100 REM RECURSIVIDAD EN BASIC
110 GOTO 500:REM INICIO MAIN
120 REM SUBRUTINA RECURSIVA
```

```
130 IF I=FINAL THEN 150
140 I=1+:PRINT "EVOCACION NUMERO ";I:GOSUB 120
150 PRINT:PRINT "¡RECURSIVIDAD,SEÑORES! ";I
160 RETURN
500 REM MAIN
510 INPUT "INDICAME EL FINAL",FINAL
520 GOSUB 120
530 END
```

Un basicalista distraído al que hiciéramos de pronto esta pregunta, seguramente contestará que el programa, por ejemplo para FIN=5, escribe:

EVOCACION NUMERO 1
EVOCACION NUMERO 2
....
EVOCACION NUMERO 5
¡RECURSIVIDAD, SEÑORES! 5

En cambio, muy sorprendido, verá aparecer el mensaje ¡RECURSIVIDAD, SEÑORES! todavía cinco veces. Lo extraño es que por un lado se tiene una interlínea y, además, no se imprimen los valores decrecientes de I desde 5 hasta 1 como quizá esperaría un pascaliano distraído, sino repetidamente:

¡RECURSIVIDAD, SEÑORES! 5
¡RECURSIVIDAD, SEÑORES! 5
..........(etc.)..........

Dado que también en BASIC (e incluso en lenguaje máquina) las subrutinas se basan en una pila que acumula los valores de retorno, el primer punto se explica en seguida: si las operaciones en reserva son las de la línea 150, también forma parte de ellas el primer PRINT y por eso se repite. En cuanto a la monotonía del valor 5 del ejemplo, se debe a que en BASIC no se tienen parámetros y todas las variables son globales.

Así comprendemos finalmente cómo puede ser ventajosa, aunque pesada a veces, la distinción entre variables de diferente nivel que nos pareció tan pedante.

Por último, hay que probar dando a FINAL valores cada vez más elevados: nos daremos cuenta de que esta recursividad del BASIC es verdaderamente espartana (el mensaje de saturación de capacidad sale para un FINAL de 20 ó 30 y las cosas empeoran trágicamente con programas largos).

Si quiere se puede construir artesanalmente una pila en un programa BASIC, de forma que se puedan reservar todas las variables y parámetros locales.

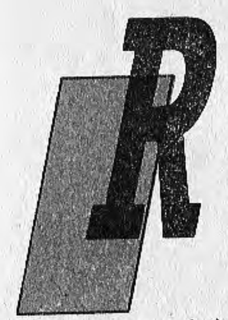
Es posible, pero es mucho más cómodo tenerlo ya todo hecho de antemano.

Como en Pascal.

CAPITULO VI

ESTRUCTURAS DE DATOS Y ALGORITMOS: TODO GUARDA RELACION

Tipos definidos por el usuario



Retomamos ahora el tema de la estructura de los datos que ya hemos visto a grandes rasgos en los primeros capítulos. Por cuestiones de espacio tendremos que esquematizar mucho este tema. Por otra parte, esto es un ensayo (pequeño) y no un tratado. Para aquellos a quienes aburre la teoría queremos recordar que sin la abstracción no se pueden dar pasos adelante ni siquiera en la práctica. De todas formas, además de hacer un rápido resumen sobre la estructura de datos del Pascal, pondremos ejemplos ilustrativos (incluso con sus correspondientes programas equivalentes en BASIC). Así trataremos de dar tambien respuesta al angustioso (y por ahora no resuelto) problema del planteamiento estructuralista de un lenguaje popular que carece de formas estructuradas. Entiéndase bien que hablamos de dialectos comunes, en los cuales no hay prácticamente ni rastro de tipos estructurados. Aludiremos fugazmente también a nuevos dialectos algo estructurados, como el SUPERBASIC, lanzado por Sinclair en su QL.

Para comodidad del lector hemos repetido en la figura 1 el cuadro sinóptico presentado en el capítulo 2. Los estándar, es decir, aquellos que vienen dados por el lenguaje, los damos ya por conocidos, pues los hemos ido presentando a medida que que surgía la ocasión. Junto a INTEGER, REAL, CHAR y BOOLEAN, hemos visto también el tipo STRING ofrecido en el UCSD Pascal por Bowles (más generoso que el "tacaño" Niklaus Wirth).

Según la clasificación de la figura 1, los estándar pertenecen

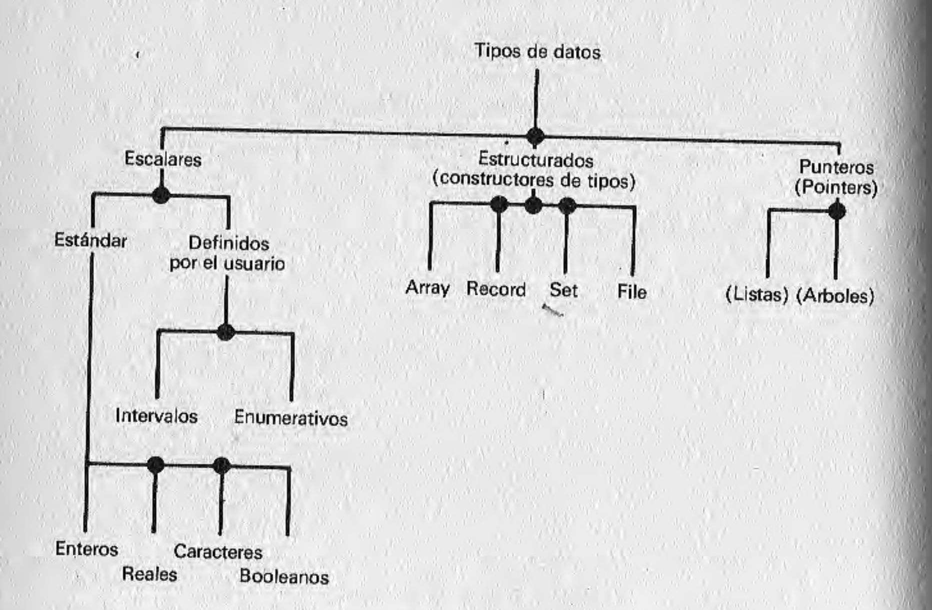


Figura 1.—Cuadro sinóptico de los tipos proporcionados por el Pascal.

a los llamados tipos "escalares". Esta denominación deriva del hecho de que pueden asumir valores únicos que forman una especie de escala de valores, en fila uno tras otro. Si lo pensamos bien, esta "granularidad" de los datos escalares no se adapta, a primera vista, a los números reales, que forman un dominio continuo no numerable; pero estas sutilezas no tienen sentido en la aritmética de precisión finita del calculador, donde también los números reales están formados por un número máximo de cifras.

Un amigo nuestro ha definido los tipos escalares definidos por el usuario como "la cenicienta del Pascal", pero su utilización inteligente permite una gran claridad en los programas y alguna ventaja inesperada.

Estos tipos escalares descritos por el usuario se definían de dos formas: por intervalo (como subconjunto de otro tipo, excluidos los reales) o bien por enumeración y que, una vez definidos, nos permiten disfrutar de:

- las funciones estándar SUCC (x), PRED (x) y ORD (x) que proporcionan, respectivamente, el elemento sucesor, o el predecesor de un dato "x", o su número de orden;
- las relaciones >,< y =.

He aquí un ejemplo instructivo. Se refiere al problema del transbordo en una barca por un campesino de un lobo, una cabra y una coliflor. Esta es la solución que proponemos:

```
PROGRAM CABRALOBOCOLIFLOR;
TYPE PERSONAJES=(CAMPESINO,LOBO,CABRA,COLIFLOR);
     PASAJERO=(LOBO..COLIFLOR);
VAR PAS1,PAS2:PASAJERO;
......
IF (ORD(PAS1))=(ORD(PAS2)+1) THEN COMILONA
......
```

El ejemplo ilustra tanto la definición por enumeración (en el TYPE personajes) como la de intervalo (en el TYPE pasajero).

El problema de este programa es que se tiene que valorar en un determinado momento, si existe el riesgo de que se active un procedimiento (genérico e intuible) llamado Comilona. Ahora bien, la condición de IF se adapta perfectamente. Tanto al lobo que se come a la cabra, como a que ésta se coma la coliflor sin necesidad de utilizar IF anidados ni CASE: ubi maior minor editur (el pez grande se come al chico).

El tipo ARRAY

Con este tipo, llamado a veces "vector", ya que sólo tiene una dimensión, se puede hacer casi todo en BASIC (en particular se consigue emular estructuras como las listas y los árboles, según veremos seguidamente). Para ser breves y suponiendo que tratamos con personas que saben BASIC (si no basta acudir a otros volúmenes de la BBI) daremos por conocidas las nociones sobre dimensión e índice. A este último, en Pascal se le llama también "selector", va colocado entre llaves cuadradas [ ] y es modificable.

A diferencia de los locos de la electrónica, que conciben un array como un conjunto de celdillas colocadas en la memoria, los estructuralistas pascalianos lo ven como un tipo abstracto asimilable al concepto matemático de matriz. Veamos la definición genérica de un tipo-array en Pascal:

TYPE Reparto = ARRAY [Ind1, Ind2...] OF Tipo.

Tipo puede ser a su vez de cualquier tipo (escalar, estructurado, estándar o creado por nosotros) mientras que Ind1, Ind2.. tienen que ser escalares. Esto permite una variedad de combinaciones impensables en el BASIC.

```
PROGRAM DIETARIO;
TYPE MESES=(ENER,FEB,MARZ,ABRI,MAY,JUN,JUL,
         AGO,SEPT,OCT,NOV,DIC);
     DINEROMENS=ARRAY[MESES] OF INTEGER;
VAR MES:MESES;ENTRADAS,SALIDAS:DINEROMENS;
............
SALIDAS[DIC]:=SALIDAS[DIC]+PAGAEXTRA;
SALDO:=0;
FOR MES:=ENER TO DIC DO
  BEGIN
   ENTRADAS[MES]:=ENTRADAS[MES]-SALIDAS[MES];
   SALDO:=SALDO+ENTRADAS[MES]
END;
WRITELN ('BALANCE ANUAL:',SALDO)
```

Hay que reconocer que, una vez definido el ARRAY dineromens que tiene como índice al tipo-escalar mes, es bien sencillo manipular, con la variable mes, el ciclo FOR para calcular el hipotético balance anual.

Lo fundamental es que el Pascal permite para los arrays multidimensionales una representación más elocuente que la DIM X(10, 20, 10) del BASIC.

Para hacer entender mejor la riqueza de expresión que se puede conseguir con el tipo ARRAY, haremos un ejemplo, parecido al anterior, pero suponiendo gastos y beneficios divididos en tres géneros: compensaciones, mercancías y servicios.

```
PROGRAM DIETARIO;
TYPE MESES=(ENER,FEB,MARZ,ABRI,MAY,JUN,JUL,
         AGO,SEPT,OCT,NOV,DIC);
     GENEROS=(COMPENSACIONES,MERCANCIAS,SERVICIOS);
     DINEROMENS=ARRAY[MESES] OF INTEGER;
VAR MES:MESES;VOC:GENEROS;ENTRADAS,SALIDAS:DINEROMENS;
........
```

Quien lo prefiera puede disponer también de la fórmula habitual. En el caso anterior sería:

```
DINEROMENS=ARRAY[MESES,GENEROS] OF INTEGER;
```

Ahora, la parte de cálculo puede estar formada por dos bucles FOR anidados uno en el otro: el interior puede totalizar sobre los tres géneros.


```
............
SALIDAS[DIC,COMPENSACIONES]:=SALIDAS[DIC,COMPENSACIONES]+PAGAEXTRA;
SALDO:=0;
FOR MES:=ENER TO DIC DO
  FOR VOC:=COMPENSACIONES TO SERVICIOS DO
  BEGIN
   ENTRADAS[MES,VOC]:=ENTRADAS[MES,VOC]-SALIDAS[MES,VOC];
   SALDO:=SALDO+ENTRADAS[MES,VOC]
END;
WRITELN ('BALANCE ANUAL:',SALDO)
```

Nos hemos alargado un poco sobre estas interesantes peculiaridades. Como ya hemos dicho, los índices son escalares, generalmente enteros, y para establecer las dimensiones se puede recurrir al criterio del rango, que consiste en separar con puntitos los extremos del campo al que pertenecen, por ejemplo:

```
VECTOR=ARRAY[100..1000] OF INTEGER
```

Se aprecia en seguida la mayor flexibilidad con respecto al BASIC, donde los índices van siempre de 0 al valor definido por nosotros.

Veamos ahora otro caso donde una estructura adecuada nos sirve para aclarar la resolución de un algoritmo y, sobre todo, su implementación. Recordamos nuevamente que los conceptos de interrelación entre algoritmos y estructuras serán tratados, sencillamente, en el volumen "Programando como es debido". Es conveniente que intente desde ahora redactar el programa equivalente en BASIC. El algoritmo que vamos a ver es la clásica "criba de Eratóstenes", que consite en suponer primero primos todos los enteros del campo a examinar, después de lo cual se eliminan todos los múltiplos de dos, luego todos los múltiplos de tres, y así sucesivamente. Si queremos trabajar con un array, nos damos cuenta, después de algunas reflexiones, de que la estructura de datos más adecuada para este sencillo algoritmo no es un array de enteros, sino un array de booleanos establecidos inicialmente como TRUE y que se van haciendo FALSE poco a poco con la técnica de la criba (ver listado de la figura 2).

Siguiendo el listado se descubre otra cuestión: los índices, tanto en Pascal como en BASIC, son variables ellos mismos y, en casos como estos, pueden ser manejados como tales. De hecho, cuando acaba el proceso del ejemplo se imprimen como números primos sólo los índices correspondientes a los booleanos correctos en el array de criba.

Los menos distraídos se darán cuenta de que esta vez no he-

```
PROGRAM Criba ;
CONS campo = 10000;
VAR hipotprim: ARRAY (2..campo) OF BOOLEAN;
    primo, pr, incr, cuentaprimos: INTEGER;
PROCEDURE todosprimos;
VAR ind: INTEGER;
BEGIN
  FOR ind:= 2 TO campo DO
    BEGIN hipotprim (ind):=TRUE  END
  END; (*Todos primos*)
BEGIN (*Main*)
todosprimos;
FOR primo:=2 TO campo DIV 2 DO
  IF hipotprim(primo) THEN
    BEGIN
    incr:=primo; pr:=primo+incr ;
      REPEAT
        hipotprim(pr):=FALSE;
        pr:= pr+incr;
      UNTIL prcampo;
    END;
WRITE ('Lista primos en el campo");
WRITELN ("de 2 a ", campo); WRITELN;
cuentaprimos:=0;
FOR primo:=2 TO campo DO
  BEGIN
    IF hipotprim(primo) THEN
    BEGIN
      cuentaprimos:=cuentaprimos+1;
      WRITELN(primo);
    END;
  END;
  WRITELN;
  WRITE(" Total primos: ", cuentaprimos);
END.
```

Figura 2.—Programa que realiza el algoritmo de la criba, de Eratóstenes. La estructura de datos que mejor se le adapta es un array de valores booleanos, al principio todos TRUE y cambiados progresivamente por valores FALSE. Al acabar son números primos los índices correspondientes a los valores del array "hipotprim" que han permanecido en TRUE.

mos introducido ARRAY como simple definidor de una VAR. En efecto, ARRAY tiene dos caras: define a una VAR de tipo array, al mismo tiempo que sirve también como constructor de tipo. Generalmente nos convendrá definir un TYPE (estructurado), ya que resulta bastante complejo cuando se tienen varias VAR del mismo tipo.

RECORD, SET y FILE. Los otros tipos estructurados

El tipo RECORD ha sido considerado por Wirth como "quizá el más flexible de los tipos de dato". Históricamente, nació con las tarjetas perforadas, subdivididas en diferentes campos, cada uno de los cuales contiene un dato de distinta naturaleza, pero que permite identificar un determinado sujeto. La sintaxis en Pascal es la siguiente:

```
TYPE <tiporecord>=RECORD <campos> END
```

Dos ejemplos típicos: 1) La tarjeta de registro que contiene la serie ordenada de apellidos, nombre, lugar de nacimiento, dirección, etc., y 2) una entrada cualquiera de una tabla que asocia el correspondiente precio (o descuento o cualquier otra cosa que se quiera) al código (o nombre) del artículo. Generalmente, el prestigio de los lenguajes estructurados es debido al hecho de que proporcionan una visión abstracta que en los lenguajes tradicionales se solía relegar a los registros (récords) situados en soportes externos. De todas formas, el BASIC no ofrece este tipo de ventajas y el programador tiene que arreglárselas con los tipos ordinarios.

Fijémonos, por ejemplo, en una tabla de descuentos. En BASIC tendríamos que apañarnos con dos arrays, digamos NOMART (NOMbre ARTículo) y DESC, que luego tendremos que hacer marchar juntos bajo nuestra personal supervisión (nada difícil, pero aburrido y poco operativo). En cambio en Pascal sería:

```
TYPE tabladescuentos=RECORD
                     nomart:STRING;
                     descuento:INTEGER
                     END;
```

En suma, se trata de una lista de datos, de distintos tipos, encerrados entre las palabras reservadas RECORD y END. Otro ejemplo clásico:

```
PROGRAM numeroscompl;
TYPE complej=RECORD rea,imag:REAL END
VAR x,y,z:complej;
PROCEDURE sumacompl(VAR sumacompl:complj;a,b:complj);
 BEGIN
  sumacompl.rea:=a.rea+b.rea;
  sumacompl.imag:=a.imag+b.imag
 END;
PROCEDURE procdcompl(VAR procdcompl:complej;a,b:complej);
 BEGIN
   procdcompl.rea:=a.rea*b.rea+a.imag*b.imag;
   procdcompl.imag:=a.rea*b.imag+a.imag*b.rea;
 END;
........
BEGIN (* main *)
 WRITELN ('dame 2 numeros complejos ');
 WRITE ('(primero la parte real,luego la imaginaria)');
 READLN (x.rea,x.imag,y.rea,y.imag);
 sumacompl(z,x,y);WRITELN;
 WRITELN ('su suma vale: ',z.rea,' +j ',z.imag);
 prodcompl(z,x,y);WRITELN;
 WRITELN ('su producto vale: ',z.rea,' +j ',z.imag);
 ..... etc......
```

El ejemplo "habla" por sí mismo. Es interesante señalar, sin embargo, que al campo de un registro (record) se accede poniendo el nombre del registro seguido por un punto y el nombre del campo. Nótese también que usamos un PROCEDIMIENTO; en efecto, en Pascal no se admiten FUNCTIONS que no sean de tipo escalar.

Son interesantes las combinaciones de este tipo de datos con otros, especialmente con el tipo array. Antes de poner un ejemplo de esto debemos referirnos al llamado RECORD "con variantes". En esencia, se trata de un RECORD (registro) que contiene por lo menos un campo que condiciona, según sus posibles valores (un número prefijado y limitado) el significado de otros campos. Por ejemplo:

```
TYPE mes=(ener,feb,marz,abril,may,jun,jul,
        ago,sept,oct,nov,dic);
     fecha=RECORD dia:1..31;mes:mes;anno:INTEGER END
     categart=(confec,agranel,deteriorable);
     producto=RECORD
```



```
               descrip:ARRAY[1..20] OF CHAR;
               almac:RECORD
                     codigma:INTEGER;
                     nombrema:ARRAY[1..10] OF CHAR;
                     END;
               CASE cat:categart OF
                 confec:(deposito:INTEGER);
                 agranel:(deposito,decim:INTEGER;um:CHAR);
          deteriorable:(deposito:INTEGER;estropeada:BOOLEAN)
     END;
```

Este ejemplo debería de ser suficientemente claro (vea lo dicho en el capítulo 2). Nos permite apreciar cómo la estructura RECORD admite cómodas subdivisiones en subcampos. En cuanto a CASE que aquí está aplicada a las "variantes", consiente campos de significado y estructura altenativa: en el caso específico de que el campo discriminador "cat" contenga "confec" irá seguido de un campo "depósito" de tipo entero; si, en cambio, son artículos a granel, le seguirán dos campos enteros ("depósito" y "decim") y un tercer campo de cadena con la unidad de medida (abreviada), mientras que en la categoría "deteriorable" se supone que "depósito" va seguido por un campo booleano que advierte si está o no estropeada (tenga en cuenta que esto es sólo un ejemplo...)

He aquí tres posibles asignaciones significativas: una hipotética VAR prod del tipo "producto" recién definido:

prod.almac.codigma: = 123
prod.cat: = confec
prod.chaqueta: = 150.000.

Por último, se podría imaginar un ARRAY de elementos del tipo producto:

```
VAR inventar:ARRAY[1..200] OF producto
```

y localizar con inventar [i]. depósito (o parecidos) un campo del i-ésimo producto

Veamos rápidamente tipo SET (conjunto) que permite formar conjuntos de datos. La sintaxis usa las palabras reservadas SET OF seguidas por un tipo que en Pascal sólo puede ser escalar o sub-rango.

Una vez definido un tipo SET se pueden hacer en él una serie de operaciones de unión de conjuntos (+), intersección (*), sustracción (–, consistente en quitar del primero todos los elementos que también son del segundo). Las comparaciones, además de

con = y <> (cuyo significado es obvio), se pueden hacer con <= o con >=, que significan inclusión. Veamos un ejemplo:

```
TYPE personajes=(lobo,cabra,coliflor,campesino);
     brigada=SET OF personajes;
VAR barca,orillizq,orillder:brigada;
........
orillizq:=orillizq+barca;
IF (lobo IN orillizq) AND (cabra IN orillizq) THEN ...
```

El programa es capaz de ofrecernos el momento en que al arribar la barca a la orilla izquierda se puede producir una situación delicada simulada por la operación unión y el sucesivo análisis de la situación; (lobo IN orillizq) restituye el valor booleano TRUE si en el conjunto "orillizq" de "brigada" se encuentra el elemento 'lobo' y lo mismo con (cabra IN orillizq). Si ambas son ciertas THEN las consecuencias son intuibles...

La instrucción WITH sirve, en la fase de elaboración, para manejar cómodamente los registros.

Prácticamente no vamos a hablar del tipo FILE (flujo secuencial); sólo confirmarles que corresponde exactamente a lo que parece y recomendarles manuales más especializados. Por otro lado nos urge tratar, aunque sea rápidamente, organizaciones más sofisticadas de datos que sólo se pueden manejar con comodidad utilizando los lenguajes estructurados.

Las estructuras dinámicas: pila (stack) y demás

Al tratar los distintos tipos pascalianos hemos pasado por alto algunas cosas. En compensación vamos a hablar ahora, de forma abreviada, del tema quizá más delicado (¿después de la recursividad? Depende de los gustos...) Por lo tanto, prepárense para utilizar su materia gris.

Hasta hace un momento hemos estado viendo estructuras "estáticas", es decir, datos organizados de una forma más o menos compleja, pero que no cambia en el curso del programa. Las estructuras dinámicas, que iremos viendo en este y los siguientes apartados, pueden ser: pilas (stack), códigos (code), listas y árboles; estos elementos hacen la delicia de los informáticos que proyectan compiladores e intérpretes, o incluso de aquellos que se ocupan de la Inteligencia Artificial.

Las pilas ya las habíamos visto cuando tratamos la recursividad, apuntando el hecho de que el stack tiene una importancia vital en el Pascal. También dijimos que los mismos microprocesa-

dores adoptan una pila para guardar en ella el estado de sus propios registros internos y la dirección de entrada de las subrutinas llamadas (una vez que se sale de ellas, es fácil recuperar todo lo que se había guardado temporalmente en la pila).

Repetimos que una pila se define intuitivamente como un montón de elementos de los que sólo el último es "visible". En inglés, una estructura como ésta es llamada "memoria LIFO" (Last In First Out), último en entrar primero en salir, dado que siempre se extrae en primer lugar el último elemento almacenado (que estaba, por tanto, en la cima de la pila).

Desde un punto de vista más riguroso, una pila consiste en una estructura de elementos homogéneos sobre la que se definen dos operaciones fundamentales: PUSH (literalmente "empuja") y POP (extrae), la una contraria a la otra. Con la primera se carga un elemento (o cualquier estructura, ya que a su vez podría estar compuesto de más elementos, como un array o un record) en la parte superior de la pila. Generalmente se le añade una estructura llamada EMPTY que sirve para indicar si la pila está vacía o, si lo prefiere, "desinflada".

El mecanismo de la pila es particularmente cómodo para el cálculo algebraico basado en la llamada notación polaca inversa (RPN - Reverse Polish Notation, utilizada, por ejemplo, en las calculadoras H. P.). Con la RPN, una expresión de tipo (a + b)*(c - d) se escribiría:

a↑b + c↑d - *

que no necesita paréntesis. En efecto, quedan sobreentendidas (como bien saben los usuarios de calculadoras de bolsillo Hewlett Packard) las dos reglas siguientes:

- al encontrar un dato lo cargamos en la pila (operación PUSH);
- si se encuentra un operador aritmético se ejecuta la operación correspondiente usando el dato situado en la cima de la pila (top) y aquel inmediatamente por debajo, consiguiente operación POP.

En la expresión vista antes, la sucesión de acontecimientos se corresponde con la secuencia ilustrada en la figura 3.

En este mismo mecanismo de la RPN se basa el lenguaje FORTH (un próximo volumen de la BBI permitirá a los curiosos profundizar en él).

Sería interesante (pero lo dejamos como ejercicio para los más avanzados) emular en Pascal o BASIC un sistema de cálculo similar. En cambio, nos tendremos que contentar con ver cómo se

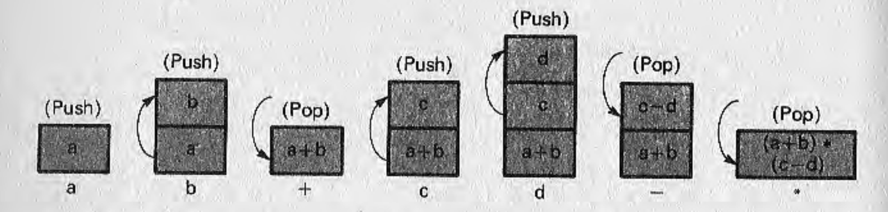


Figura 3.—Sucesivos PUSH y POP de la pila necesarios en el cálculo de una fórmula aritmética con la notación polaca inversa (RPN).

podría simular una pila en Pascal y en BASIC. En BASIC esto es indispensable en todos los casos prácticos, mientras el Pascal, que posee su propia pila, no nos permite construir, mediante una definición, un tipo-pila. Por lo tanto, nos tendremos que arreglar, en ambos casos, con un array.

La realización se muestra en la figura 4. Como se puede ver, el puntero "punt" (en BASIC llamado PT) es, en realidad, un vulgar índice que, en el procedimiento PUSH, es incrementado antes de cargar el siguiente elemento del array, mientras que en el POP primero se extrae el elemento y después se decrementa punt. En los dos ejemplos hay características no necesariamente universales, como la elección de elementos de tipo cadena en el array (la cadena, única riqueza verdadera del BASIC, puede resultar cómoda para cargar grupos de datos, pero en Pascal puede ser preferible el tipo-record) o el uso de una FUNCTION para el POP (alguien podría preferir que una misma variable "z" sirviese como bandera de carga para PUSH o de descarga para POP). Otra particularidad es haber escogido la función Offlimits para señalar cualquier condición anómala del índice (a otros les habría gustado más que se distinguiera entre la situación de "empty" —vacía— y la de "overflow" —desbordamiento—).

En cuanto al programa BASIC, el paralelismo con el Pascal nos ha estimulado a utilizar DEF FN, llegando al punto de poner el main debajo, precedido por la subrutina.

Un paréntesis: el dilema bottom-up/top-down

Al habernos centrado en la creación de programas para el manejo de la pila (stack) reutilizables en otro lugar, hemos dejado suelto el MAIN, constituido por un vacío encerrado entre los terminadores BEGIN y END. Esto nos ofrece la oportunidad de hacer una observación: no es del todo cierto que el Pascal se ciña exclusivamente a un planteamiento bottom-up (de abajo a arriba). Si


```
PROGRAM emuladorpila;
  CONST maxpun:=200;
  VAR pila: ARRAY(1..maxpunt) OF STRING;
      punt: INTEGER;

PROCEDURE pricpila;
  BEGIN punt:=0;
  END;

FUNCTION offlimits: BOOLEAN;
  BEGIN
    IF punt=maxpunt THEN
      offlimits:=TRUE;
    ELSE offlimits:=FALSE;
  END;

PROCEDURE push(x:STRING);
  BEGIN
    IF NOT offlimits THEN
      BEGIN
        punt:=punt+1; pila(punt):=x
      END;
  END;

FUNCTION pop:STRING;
  BEGIN
    IF NOT offlimits THEN
    BEGIN
      pop:=pila(punt); punt:=punt-1;
    END;
END;

BEGIN (*Main*)
..
..
END.
```

```
100  MAXPT=200: DIM PILA$(MAXPT)
110  GOTO 500: REM SALTO AL INICIO DEL MAIN
120  DEF FN OFFLIMIT=(PT=0) OR (PT=MAXPT)
130  REM PUSH
140  IF OFFLIMIT THEN RETURN
150  PT=PT+1: PILA$=X$
160  RETURN
170  REM POP
180  IF OFFLIMIT THEN RETURN
190  POP=PILA$(PT): PT=PT-1
200  RETURN
500  REM INICIO DEL MAIN
```

Figura 4.—Emulación de la estructura de pila: a) en Pascal y b) en BASIC.

lo desea, puede utilizar el top-down (de arriba a abajo). Esta es la demostración:

```
PROGRAM arribaabajo;
VAR lo,que,te,parezca:STRING;
FUNCTION condic:BOOLEAN;
 BEGIN
  READLN (lo)
  IF lo='preg' THEN condic:=TRUE
   ELSE condic:=FALSE
 END;
PROCEDURE loseyo;
BEGIN
END;
PROCEDURE losabestu;
BEGIN
END;
PROCEDURE mannana  (* no se puede usar ñ *)
BEGIN
END;
FUNCTION pasadomannana;
END;
BEGIN (* main *)
loseyo;losabestu;
IF condic THEN mannana
 ELSE pasadomannana
END.
```

El ejemplo es humorístico, pero pretende exponer cómo si determinamos con precisión la forma exacta en que tendrá que comportarse el main y las piezas que lo componen (funciones y procedimientos), será preciso, ante todo, el main mismo, mientras que sus módulos podrán estar vacíos. Dado que el compilador permanece indiferente ante procedimientos y funciones vacías, formadas sólo por parejas BEGIN/END, se podrá expurgar exclusivamente el main. Luego haremos los refinamientos paso a paso, definiendo los diferentes módulos, con sus correspondientes comprobaciones, según se vaya completando el esquema.

Como se puede intuir con el ejemplo, el desarrollo no es, en absoluto, banal: ha sido necesario insertar la función Condic para emular, mediante el teclado, un hecho que proporcionará alternativas. Si todos los main se dejasen reducir a simples secuencias de procedimientos, inicialmente vacíos, sería estupendo, pero, por

desgracia, no suele pasar. En suma, en los casos reales la conexión entre módulos y programas principales es muy estrecha y el acercamiento progresivo del planteamiento top-down sólo contempla un aspecto de la programación estructurada.

Antes de acabar con las pilas debemos referirnos a otra estructura: la cola (queue en francés), que también aparece bajo las siglas FIFO (First In First out, primero en entrar, primero en salir). Las funciones básicas de una cola (o lista de espera) se pueden definir como "pon en la cola" ("coloca") y "saca" ("sirve"). Se utiliza en los sistemas operativos para gestionar los almacenamientos intermedios de entrada/salida (donde se guardan los datos transmitidos o recibidos), y en los sistemas multiusuario para gestionar ecuánimemente las solicitudes de acceso (a impresora, disco rígido, etc.); también se podrían utilizar en un programa de simulación de hechos basados en la teoría de las colas (en estadística, por ejemplo).

La emulación de una cola, con el uso del habitual y servicial array, no es muy difícil. Bastará con dos índices-punteros, el primero para la colocación de datos en la cima y el segundo para retirar los del final; esta vez ambas operaciones incrementan los punteros. Esto provoca un deslizamiento continuo hacia la parte alta de la memoria, que hace indispensable disponer de una organización circular para el array. No se asusten, que no es mucho: bastara con obligar a la re-inicialización del puntero cuando llegue al valor máximo, mediante instrucciones del tipo:

```
BEGIN
  puntuac:=puntuac+1;
  IF puntuac=puntuacmax THEN
   puntuac:=0
  .....
```

Listas lineales

Desde un punto de vista exclusivamete lógico, una lista lineal L es un conjunto de elementos dispuestos consecutivamente. Generalmente se definen las funciones siguientes (con estos u otros nombres):

- Primero (L), que nos da el primer elemento de L;
- continuación (L), que nos da lo que queda de L después de haber quitado el que era primer elemento;
- inserción de un nuevo elemento entre dos elementos cualquiera;
- supresión de un elemento cualquiera.

No se trata, como en el caso de las matrices (arrays) de estructuras de acceso directo, pero de todas formas su carácter dinámico permite un manejo fácil, basado en una modificación continua de los elementos que forman parte del conjunto.

El punto característico de las listas (y, como se verá, de los árboles) es que no se trabaja con índices, sino con *apuntadores*, es decir, con parámetros, que, normalmente, forman parte del elemento y que "apuntan" al siguiente. Se habla de estructuras en "cadena". En la figura 5 se pone en evidencia el juego de apuntadores necesario para insertar y para suprimir un elemento.

El típico ejemplo de lista lineal es un texto, entendido como un conjunto de líneas (o palabras) en el que se pueden insertar o suprimir líneas (o palabras) en el que se pueden insertar o suprimir líneas (o palabras). Análogamente, los sectores de un disco a

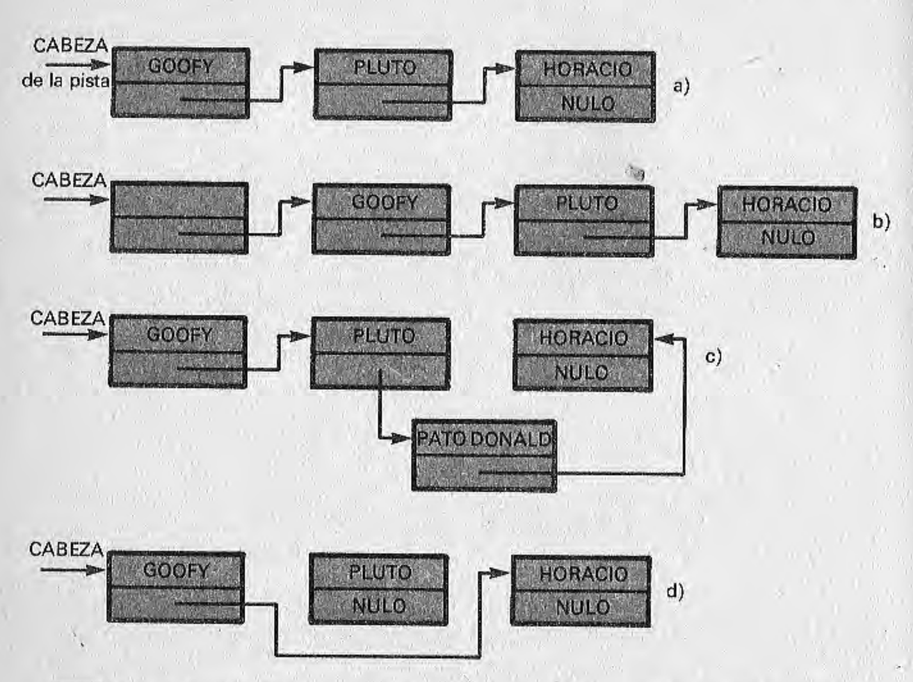


Figura 5.—a) Esquema de la organización de una lista lineal. Por norma, el primer puntero mira hacia la "cabeza" de la lista. En b) se ilustra la inserción de un nuevo elemento. Manejando adecuadamente los punteros se puede insertar un nuevo elemento en cualquier sitio. c) Por último, en d) tenemos la eliminación de un elemento: es suficiente con cambiar un puntero.

menudo están dispuestos en forma de lista, para permitir una gestión dinámica del espacio.

En los lenguajes estructurados como el Pascal (por no hablar de Modula 2 o Ada) cualquier elemento de una lista está compuesto por dos campos: el campo del valor y el campo del apuntador (pointer). El apuntador es un tipo muy abstracto. En efecto, no se dice cómo está representado "materialmente": ¿será un índice entero?, ¿un código? No se sabe: depende de la implementación. De todas formas, no está de más saber que, en la totalidad de los casos prácticos, se trata de la dirección del dato hacia el cual "apunta". En Pascal se usa la siguiente definición:

```
VAR apunt:↑elemento;
```

donde la flecha (que también es el símbolo de exponenciación, a menudo sustituido por el acento circunflejo "^") indica que "apunt" es un tipo-pointer que apunta hacia "elemento". Un valor particular del apuntador es NULO (NIL), que significa que apunta... hacia ningún sitio. Sirve para las listas vacías o para señalar el final (el último elemento tiene un apuntador NULO). Veamos cómo se puede definir (recursivamente) una lista de palabras:

```
TYPE apunt:↑elemento;
     elemento=RECORD
              valor:STRING;
              continuacion:↑elemento;
              END;
```

Vale la pena observar que éste es el único caso del Pascal en el que un "elemento" es referido antes de que esté completa su definición. El tipo "lista" está definido como un apuntador hacia "elemento" que, a su vez, es un registro formado por los campos "valor" y "continuación" —que apunta al siguiente— (obviamente los campos de datos podrían ser más de 1, lo mismo que los apuntadores para listas más complejas o árboles).

En la figura 6 tenemos: el listado del programa que contiene procedimientos en Pascal para el manejo de una lista lineal de caracteres (6a) y su equivalente en BASIC (6b); en el segundo caso nos las hemos tenido que arreglar por fuerza con dos vectores, el de los valores y el de los apuntadores.

En vista de que estos temas serán más detalladamente tratados en un próximo libro referente al manejo de datos, nos limitamos a dos notas indispensables sobre el tipo pointer:

- un elemento de la lista se especifica con una instrucción del tipo pt↑.valor;

```
PROGRAM listalineal;
TYPE lista=↑elemento;
     elemento=RECORD
              valor:CHAR;
              contin:lista
              END;
FUNCTION primero(a:lista):CHAR;
 BEGIN
  primero:=a↑.valor;
 END
FUNCTION sucesor(a:lista):lista;
  BEGIN
   sucesor:=a↑.contin
  END
FUNCTION empty(a:lista):BOOLEAN;
 BEGIN
  IF a:=NIL THEN
   empty:=TRUE
  ELSE
   empty:=FALSE
 END;
FUNCTION inser(c:CHAR;a:lista):lista;
VAR pt:lista;
  BEGIN
   new(pt);
   pt↑.valor:=c;
   pt↑.contin=a;
   inser:=pt
  END;
FUNCTION busca(c:CHAR;a:lista):lista;
(* da el primer elemento que coincide con c *)
VAR enc:BOOLEAN;
 BEGIN
  enc:=FALSE;
  WHILE NOT enc AND (a<>NIL) DO
   IF c=a↑.valor THEN enc:=TRUE
   ELSE a:=contin(a);
  busca:=a
 END;
END;
BEGIN (* main *)
END.
```



```
100 DIM LISTA$(200),SEG(200):GOTO1000
150 DEF FN PRIM$(L)=LISTA$(L)
200 DEF FNSUCC(L)=SEG(L)
250 DEF FNEMPTY(L)=L=0
300 REM CREACION DE NUEVO ELEMENTO
310 REM CON RESULTADO EN X
320 LIB=LIB+1:REM NUEVO PUESTO LIBRE
330 X=LIB
340 RETURN
350 REM
390 REM EMULACION DE LA FUNCION
400 REM INSER$(C$:CADENA;L1:LISTA)
410 REM RESULTADO
420 GOSUB 300:REM EMULA LA new(pt)
430 LISTA$(X)=C$:SEG(X)=L1:L2=X
440 RETURN
450 REM
480 REM EMULAC. DE soppress(L:LISTA)
490 REM RESULTADO EN L
500 L=FNSUCC(L)
510 RETURN
520 REM
570 REM EMULAC. DE BUSCA (C$:CADENA;L1:LISTA)
580 REM RESULTADO EN L2
590 REM B=VARIABLE (PSEUDO) LOCAL
600 B=-1:L2=L1
610 IF L2<>0 AND B THEN 640
620 IF LISTA$(L2)=C$ THEN B=0:GOTO 610
630 L2=FNSUCC(L2):GOTO 610
640 RETURN
1000 REM INICIO MAIN
.......(a placer).......
```


Figura 6.—Funciones típicas para el tratamiento de una lista lineal, en Pascal (a) y en BASIC (b).

- la función new(pt) dada por el Pascal sirve para crear un apuntador pt, o sea, para dejar espacio cada vez que se inserta un nuevo elemento (así se abrirá e intercalará un espacio en memoria).

Con referencia a las funciones de la figura 6, recuerden que una lista es localizada apuntando a su primer elemento, lo que supone que, recorriéndola hacia adelante, "se pierden" los elementos barridos. Si no queremos que esto suceda, será necesario utilizar dobles apuntadores (listas bidireccionales). Finalmente, hay que decir que, en el programa de la figura 6, la función-lista Inser añade un nuevo carácter "c" a la cabeza de la lista "a", por lo que si quiere una inserción intermedia deberá usar previamente la función Busca.

Otra laguna es el borrado, pero llenarla es sencillo. Utilizando siempre un apuntador auxiliar (pt, por ejemplo), como en la inserción, la cancelación del elemento que sigue a pt se hace con una sola instrucción:

```
pt↑. contin: = pt↑. contin. contin.
```

Efectivamente, en el segundo miembro se tiene el desplazamiento de dos elementos y el apuntador resultante es asignado al primer miembro, saltándose así el elemento intermedio.

Trepar a los árboles

El ejemplo visto en la figura 6 justifica en parte el motivo que indujo a Wirth a no incluir el tipo cadena ni siquiera en el Modula 2: en efecto, una vez definida como lista de CHAR, una cadena se construye y se manipula con gran facilidad.

Otras estructuras de datos dinámicas son las listas bidireccionales y las circulares: las primeras dotadas de dos apuntadores por elemento, de los que uno apunta al sucesor y el otro al predecesor, y las segundas caracterizadas porque el apuntador del último elemento mira al primero (no hay ningún apuntador NIL). Estas listas unen a las ventajas ya vistas —borradas e inserciones fáciles— la del barrido secuencial; por ejemplo, si queremos mantener un cierto orden cada vez que se añade un elemento, en una lista ordenada según un criterio, no será necesario recorrer la lista desde el principio, sino que basta ir desde donde estemos hasta el punto en que el elemento nuevo tenga que insertarse, asignándole entonces el código intermedio correspondiente.

El árbol (tree, en inglés) nace también como estructura por esta exigencia, pero posee otras muchas ventajas y aplicaciones.

El árbol constituye un conjunto organizado en sentido jerárquico, y está generalmente "invertido", o sea, representado con el tronco para arriba: su vértice, llamado "raíz", está arriba, mientras que las ramificaciones están hacia abajo. En conjunto, un árbol es un "grafo", es decir, un conjunto de nudos unidos por ramas. De cada nudo parten ramas que se dirigen ("apuntan") a nudos de nivel inferior. Usando una terminología botánica, los nudos terminales se llaman "hojas". A veces también se usa una nomenclatura de tipo genealógico, y en este caso se hablará de nudos "padre" y nudos "hijo". Evidentemente, una estructura de datos arborescente también se puede utilizar para representar un auténtico árbol genealógico: en la figura 7 vemos el pedigree de una hipotética generación de perros de raza. Esto mismo vale para las clasificaciones por géneros o especies, en las ciencias naturales, en el archivo de una biblioteca o en un almacén de piezas de recambio.

Un típico caso informático es el constituido por la organización en árbol de los sistemas de gestión de archivos (adoptado sobre todo en el sistema operativo UNIX de AT&T, pero también en el ProDOS del Apple IIc/IIe y en el MS DOS del PC IBM y compatibles).

Seguramente es la flexibilidad de los árboles, lo que prefieran los informáticos: podemos ir desde la jerarquía top-down en los programas para valoración de los movimientos de un juego

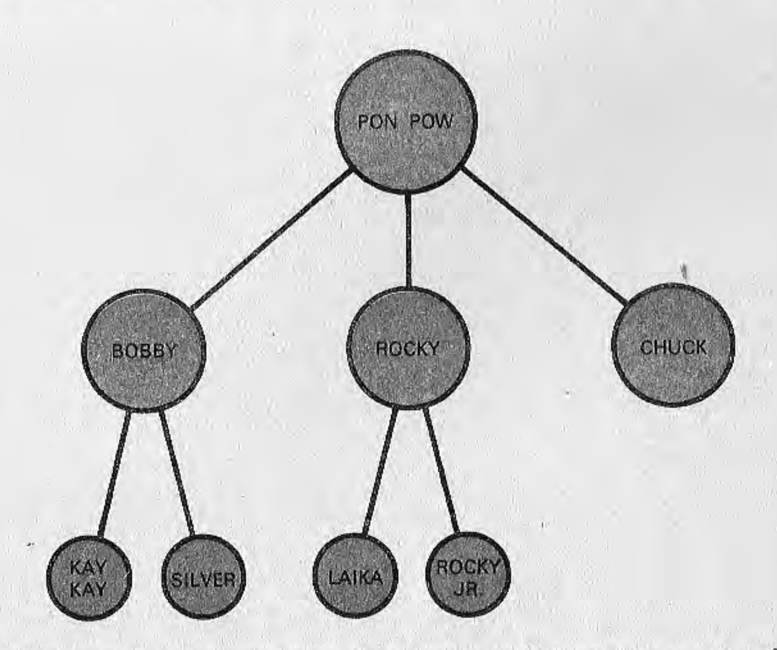


Figura 7.—Una estructura de datos en árbol invertido podría muy bien representar árboles genealógicos, como los pedigrees de los perros de raza...

(ajedrez, damas, etc.) a la arquitectura de compiladores, por no hablar de la Inteligencia Artificial, sector donde se les considera como a reyes.

En la clasificación se habla de niveles (un nudo hijo es de nivel inferior al de su padre) y de orden (número máximo permitido de ramas) definiéndose como "sub-árbol" todo árbol que tenga como raíz un nudo distinto al del cabeza de la serie (raíz principal).

Nosotros nos vamos a limitar a una categoría particular, muy importante, dentro de los árboles: los árboles binarios (aquellos en los que el orden es dos). Cada nudo tiene, por tanto, como mucho, dos hijos, distinguidos habitualmente como izquierdo y derecho. El árbol binario es útil por su mayor sencillez. Árboles con un orden mayor que dos se pueden reducir con las adecuadas reglas a árboles binarios.

Estas son las operaciones básicas que definen a un árbol binario:

- acceso,
- creación,
- verificación.

El acceso consiste en tres operaciones de lectura: raíz, hijo derecho e hijo izquierdo. La creación se realiza partiendo de dos subárboles e introduciendo una raíz nueva. La tercera función comprueba si un árbol está vacío o no.

De forma análoga a lo visto para las listas, un árbol binario se puede simular con tres arrays asociados: el vector de los valores (nudo) y los de los punteros izquierdo y derecho (ramas). En BASIC ésta es la única implementación posible, mientras que en Pascal el tipo pointer sirve mucho mejor para este fin.

Visitar un árbol

Una importante noción al emplear los árboles es la llamada técnica de la "visita", que describe las posibles modalidades para recorrer todos sus nudos. Con las listas se puede proceder sólo en dos sentidos; ahora son tres las posibilidades. Si llamamos R, I, D, respectivamente, a la raíz y a los hijos izquierdo y derecho, las secuencias de visita son:

- visita pre-orden: R-I-D,
- visita post-orden I-D-R,
- visita en-orden: I-R-D.

Refiriéndonos a la figura 8, y teniendo en cuenta que un nudo que no sea una hoja se identifica por el subárbol del que es raíz, las tres visitas dan lugar a las siguientes sucesiones:

- pre-orden: A-B-D-E-C-F-H-I-G;
- post-orden: D-E-B-H-I-F-G-C-A;
- en-orden: D-B-E-A-H-F-I-C-G.

Para entendernos mejor: en esencia por "visita" se entiende el alcanzar de forma consecutiva determinados nudos, respetando unas reglas dadas (según la técnica de visita); en algunas ocasiones será necesario, antes de llegar a un nudo en concreto, barrer en profundidad otros nudos, "dejándolos en reserva". Haciéndolo prácticamente, primero hay que salir de la raíz, después de lo cual, por ejemplo en una visita post-orden, tendremos que bajar hasta la hoja D (Fig. 8) antes de empezar; una vez recorrido el subárbol D-E-B pasaremos (¡sin "asumirla"!) por la raíz y desde ella bajaremos a H para volver a empezar la visita. "Dejar en reserva" algo supone, en este caso, la utilización de una pila (stack), posiblemente automática...

Una aplicación interesante de los árboles binarios y sus correspondientes visitas la tenemos en las expresiones algebraicas. Refiriéndonos a la figura 8b), un árbol que represente en sus hojas operadores aritméticos y datos puede dar lugar a tres notaciones:

NOTACION	VISITA	RECORRIDO
prefija	pre-orden	*+*5 3 2 – 8 4
postfija	post-orden	5 3*2 + 8 4 – *
infija	en-orden	(5*3 + 2)*(8 – 4)

La tercera es la ordinaria, mientras que la post-fija es la RPN ya vista. En la última línea hemos puesto paréntesis sólo para recordar que el resultado se va calculando sin prioridades; será 68, como en las otras dos visitas:

Recursividad: ventajas prácticas

En la figura 9 presentamos un programa en Pascal completado esta vez con un pequeño main didáctico, utilizado para demostrar las tres técnicas de visita en el caso de la expresión algebraica vista en la figura 8b). La definición del tipo algbin es evidente: con respecto a la lista, se trata solamente de un puntero más. También la FUNCTION Crea tiene una perfecta analogía con la Inser

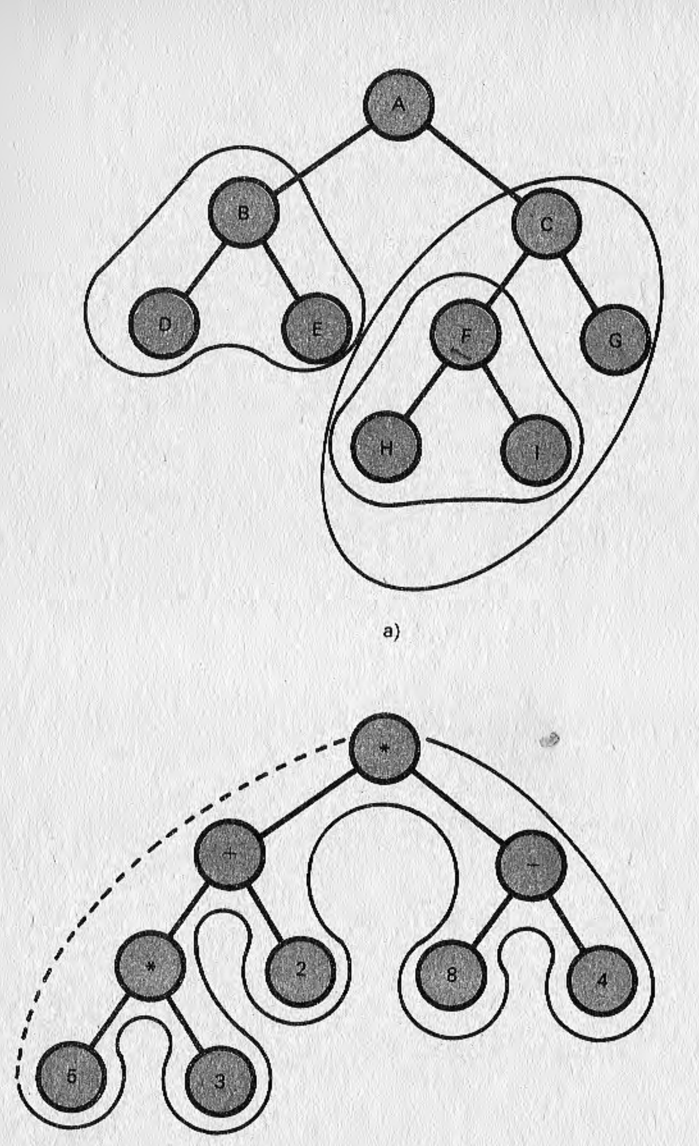


Figura 8.—En a) tenemos un árbol binario genérico y en b) el correspondiente a la representación de una expresión algebraica. En ambos casos se pueden realizar las tres técnicas base de "visita". En a) se pueden apreciar varios subárboles (de orden>1) mientras que en b) la visita en-orden está representada con líneas de trazos para las bajadas al "hijo izquierdo" y líneas continuas para la visita hijo-padre-hijo.


```
PROGRAM expresalg;
TYPE algbin=↑elemento;
     elemento=RECORD
               valor:STRING;
               hijoizq:algbin;
               hijoder:algbin;
              END;
VAR expres:algbin;
FUNCTION crea(val:STRING;hizq,hder:algbin):algbin;
VAR pt:algbin;
BEGIN
 new(pt);
 pt↑.valor:=val;
 pt↑.hijoizq:=hizq;
 pt↑.hijoder:=hder;
  crea:=pt;
END;
PROCEDURE preorden(a:algbin);
BEGIN
 IF a<>NIL THEN
  BEGIN
   WRITE (a↑.valor);
   preorden(a↑.hijoizq);
   preorden(a↑.hijoder);
  END
END;
PROCEDURE postorden(a:algbin);
BEGIN
 IF a<>NIL THEN
  BEGIN
   postorden(a↑.hijoizq);
   postorden(a↑.hijoder);
   WRITE (a↑.valor)
  END
END;
PROCEDURE enorden(a:algbin);
BEGIN
 IF a<>NIL THEN
  BEGIN
   enorden(a↑.hijoizq);
   WRITE (a↑.valor);
```

```
    enorden(a^.hijoder);
   END
  END;
  BEGIN (* main *)
   expres:=crea('*',crea('+',crea('*',
          crea('5',NIL,NIL),
          crea('3',NIL,NIL)),crea('2',NIL,NIL)),
          crea('-',crea('8',NIL,NIL),
          crea('4',nil,nil)));
   preorden(expres);WRITELN;
   postorden(expres);WRITELN;
   enorden(expres)
  END.
```

Figura 9.—Listado del programa Expresalg, que comprende la función de creación de un árbol binario y la de los tres tipos de visita. En el main se genera el árbol de una expresión algebraica, y se realiza su visita por los tres modos, imprimiendo, para su verificación, las expresiones correspondientes.

del programa de tratamiento de listas de la figura 6, aparte de tener dos apuntadores en lugar de uno. Despues de esto, sugerimos dar un vistazo al main: en la parte introductoria, donde secrea el árbol de la figura 8b), las hojas se definen en las expresiones que contienen dos apuntadores NIL, mientras que para los demás elementos Crea sigue el mecanismo de la visita pre-orden, adoptada como la más natural.

Para ayudar a comprender este intrincado parentesco, les sugerimos empezar por escribir de nuevo la visita pre-orden de la figura 8b), insertando esta vez los paréntesis necesarios para evidenciar los distintos subárboles:

* (+ (* 5 3) 2) (- 8 4)

Una vez llegados aquí, ya sólo se trata de un juego de paciencia: recurrir a la función Crea añadiendo los elementos que definen las cadenas en Pascal y las comas que separan los tres parámetros de Crea, sin olvidar todas las hojas, como 5, que son traducidas en Crea según ('5', NIL, NIL).

Notará inmediatamente la comodidad típica del Pascal en las llamadas anidadas de las funciones (sólo son necesarios unos pocos paréntesis), pero esto no es nada con respecto a la potencia expresiva de las auto-invocaciones recursivas de los tres procedimientos Preorden, Postorden e Inorden: la autoinvocación funciona, ya sea para el hijo izquierdo o el derecho, mientras que el subárbol-parámetro no sea NIL.

Tenemos que "creernos" todo lo que se ha dicho sobre la recursividad y admirar el hecho de que, en el ejemplo en cuestión, las tres técnicas de visita se diferencien sólo por la sucesión con que el WRITE (para la impresión del valor del nudo alcanzado) y las instrucciones relativas a las ramas izquierda y derecha se presentan.

Una aplicación muy interesante de los árboles binarios es con los conjuntos de datos "verdaderos y propios". El método permite mantener un ordenamiento de una forma sencilla y suficientemente veloz. El algoritmo consiste en crear una arborescencia insertando sistemáticamente los datos inferiores a la raíz en el subárbol izquierdo, y en el derecho, los datos superiores. Este proceso equivale al método de bisección (algoritmo utilizado normalmente para el ordenamiento y la búsqueda en el array): se opera primero con la mitad, luego con la mitad de la mitad y así sucesivamente, por lo que da mejores resultados que las listas, como ya hemos dicho. De todas formas, para poner simplemente un ejemplo, suponga que van llegando a un árbol binario, vacío en principio, los valores siguientes: 17, 12, 5, 80, 10, 35, 8, 6, 11, 94, 27, 24, 20, 48. Si ha entendido cómo funciona el procedimiento, intente construir el árbol binario, y después compárelo con el de la figura 10, donde al lado de cada nudo figura entre paréntesis el orden de llegada. Una pregunta (será la última): ¿cuál de los tres tipos de visita —pre-post y en-orden— proporciona una salida ordenada de los datos en un árbol binario semejante?

Veamos el listado directamente:

```
PROGRAM insercion;
TYPE algbin=^elemento;
     elemento=RECORD
              postcp:INTEGER;
              ciudad:STRING;
              hijiz,hijder:algbin;
              END;
VAR albcp:algbin;cp:INTEGER;ciu:STRING;
FUNCTION crea....
(* parecida a la de la fig.8 *)
FUNCTION enorden...
(* parecida a la de la fig.8 *)
END;
```

```
FUNCTION inser(x:INTEGER;y:STRING;a:algbin):algbin;
 BEGIN
  IF a=NIL THEN
   ins:=crea(x,y,NIL,NIL)
  ELSE
   IF x<=a↑.postcp THEN
    inser:=inser(x,y,a↑.hijiz)
   ELSE
    inser:=inser(x,y,a↑.igder)
 END;
BEGIN (* main *)
 albcp:=NIL;
 REPEAT
  READLN (cp,ciu):inser(cp,ciu,albcp);
 UNTIL cp=0
 enorden(albcp);
END.
```

Ha sido imaginada la creación con la visita de control de una base de datos de los cp (códigos postales) con las correspondientes ciudades; en consecuencia, el registro ahora contiene dos valores: cp y ciudad. Después, la parte dejada en reserva con los puntitos es prácticamente idéntica a la de la figura 9, excepto pequeñas variantes que tengan que ver con la nueva estructura del tipo algbin.

Una vez más, la FUNCTION Inser es elocuente. No es más que la traducción formalizada y recursiva del algoritmo descrito: si el árbol está vacío crea la raíz, si no Inser(ta) la nueva pareja (cp y ciudad) leída por el teclado. Advierta que, gracias al mecanismo recursivo, la condición IF también nos hace salir de Inser, por lo que el último elemento alcanzado ha encontrado un lugar. En efecto, Crea (x, y NIL, NIL) se adapta justamete a cada nuevo nudo, raíz de momento sin hijos, pero potencialmente prolífica. Esto es lo que significa una estructura dinámica.

Para terminar, la última línea del main nos revela en seguida que la visita Enorden corresponde, obviamente, a la definición misma de la base de datos ordenada por el árbol binario.

Una breve despedida

Alguien se preguntará: ¿hay un programa BASIC equivalente a éste? Desgraciadamente, esta vez no hay ni siquiera reflejos de todos los fuegos artificiales-recursivos del Pascal. Los caminos a recorrer podrían ser los siguientes:

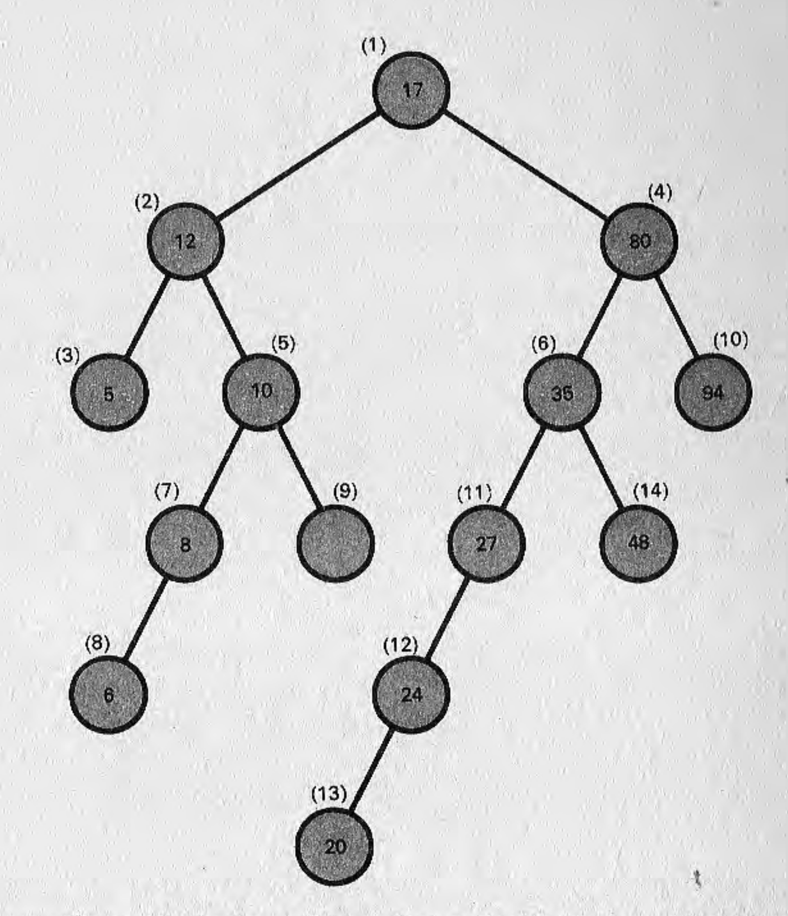


Figura 10.—La figura ilustra cómo se "cuelgan" de un árbol binario valores dispuestos en orden. Los números externos, entre paréntesis, indican el orden de llegada, según la sucesión mencionada en el texto.

- implantar, según ya hemos sugerido en otro lugar, una pseudo pila y "engancharla" a las llamadas recursivas;
- en general es más sencillo emular la recursión con la iteración, y, en este caso específico, el truco consiste, a grandes rasgos, en insertar los adecuados punteros.

Pero ambas explicaciones exceden los límites de este libro. Esperamos que aquellos que no tengan la paciencia o el tiempo necesarios para profundizar un poco más en estos temas, por lo

menos habrán aprendido algo. Queremos puntualizar algunas cuestiones antes de despedirnos:

- últimamente ha remitido un poco la polémica con los estructuralistas, e incluso, en los libros más recientes sobre lenguajes de programación, podemos encontrar cierta permisividad (en especial en el lenguaje C);
- en el duelo BASIC-Pascal todavía domina (aunque por poco) el primero por dos motivos: el primero va ligado a la mayor cantidad de programas desarrollados; el segundo es la mayor flexibilidad del BASIC, sin las obligaciones tan formalizadas del Pascal, que, a veces, pueden parecer demasiado complicadas a los profanos;
- la llegada de BASICs estructurados supondría el triunfo inmediato de los principios de la programación estructurada: por ejemplo, en el SuperBASIC usado en el QL de Sinclair, no falta prácticamente nada: construcción IF/ELSE (incluso mejor que la del Pascal, ya que tiene el terminador ENDIF, que evita ciertos problemas de los que ya hablamos a su debido tiempo), FOR y REPEAT (con EXIT anticipado de un bucle, cuya falta en Pascal es incómoda a veces) y procedimientos y funciones con variables locales y parámetros.

Sin necesidad de esperar la muerte del BASIC tradicional —siempre anunciada, pero aún lejana—, hoy en día los cánones de la programación estructurada se imponen cada vez más: la gradual inclinación hacia instrumentos potentes y expresivos es, sin duda, sincera por parte de muchos BASICalistas empedernidos.

Hablando en términos generales, esos cánones son la claridad y la precisión, pero la auténtica idea central es la íntima unión de la estructura de datos con los procedimientos (algoritmos). Desarrollarla, incluso con los pobres instrumentos de los lenguajes tradicionales, es casi un deber; ofrece grandes ventajas y, a la vez que muchos dolores de cabeza, bastantes satisfacciones.

BIBLIOGRAFIA

PASCAL a partir del BASIC
Brown. *Anaya Multimedia*

UCSD PASCAL
David Price. *Gustavo Gili*

Primeros pasos en LOGO
Otero, Pueyo, Cajaravilla. *Anaya Multimedia*

Programación en LOGO
J. D.—Opazo, 1985. *Anaya Multimedia*

El libro del IBM PC, XT, AT
L. E. Frenzel Jr./ L. E. Frenzel III, 1985 *Anaya Multimedia*

Programación en C. Introducción y conceptos avanzados
M. Waite, S. Prata, D. Martín, 1985. *Anaya Multimedia*

Informática para no avanzados.
C. Willmott, 1985. *Deusto*

Cómo programar ordenadores personales
R. Farrando, 1985. *Marcombo.*

Los ordenadores de la Quinta Generación.
G. L. Simons, 1984. *Díaz de Santos.*

Compiladores e intérpretes. Un enfoque pragmático.
G. Sánchez Dueñas, J. A. Valverde Andreu. *Díaz de Santos*

Elección y compra del software de gestión.
J. Tangui, 1985. *Deusto.*

Los ordenadores. Fundamentos y sistemas.
J. C. Giarratano, 1984. *Díaz de Santos.*

Conceptos actuales sobre la tecnología de los ordenadores.
J. C. Giarratano, 1984. *Díaz de Santos.*

Diccionario de Informática inglés-español-francés
G. A. Mania, 1985. *Paraninfo*

Diccionario del BASIC.
Willie Hart, 1985. *Paraninfo.*

Diccionario de Informática inglés-español. Glosario de términos informáticos.
Olivetti, 5.ª edic., 1984. *Paraninfo.*

Diccionario de Informática.
Masson, 2.ª edic., 1985. *Masson.*

Método general de análisis de una aplicación informática.
Tomo 2. Etapas y puntos fundamentales del análisis orgánico y...
Castellani, 1985. *Masson.*

Introducción a la programación 2. Estructuras de datos.
Clavel-Biondi, 1985. *Masson.*

BIBLIOTECA BASICA INFORMATICA

INDICE GENERAL

10 Sistemas operativos y software de base
Qué son, para qué sirven. Unos desconocidos muy importantes.
11 Sistema operativo CP/M
Uno de los sistemas operativos para microprocesadores de 8 bits de mayor difusión en el mercado.
12 MS-DOS: el estándar de IBM
Sistema operativo para el microprocesador de 16 bits 8088, adoptado por el IBM-PC.
13 Paquetes de aplicaciones. Software "pret a porter"
Características y peculiaridades de los más importantes paquetes de aplicaciones.
14 VisiCalc: una buena hoja de cálculo
Interioridades y manejo de una de las hojas de cálculo más usadas.
15 Dibujar con el ordenador
Profundizando en una de las facetas útiles y divertidas que nos ofrecen los ordenadores.
16 Tratamiento de textos... para escribir con el ordenador
Cómo convertir su ordenador en una máquina de escribir con memoria y todo tipo de posibilidades.
17 Diseño de juegos
Particularidades características de esta aplicación de los ordenadores.
18 LOGO: la tortuga inteligente
Un lenguaje conocido por su «cursor gráfico», la tortuga, y sus aplicaciones pedagógicas al alcance de su mano.
19 BASIC y tratamiento de imágenes
Todo lo que en **¡Y llegó el BASIC!** no se pudo ver sobre las imágenes y gráficos en el BASIC.
20 Bancos de datos (I)
Peculiaridades de una de las aplicaciones de los ordenadores más interesantes, y que más dinero mueven.
21 Bancos de datos (II)
Profundizando en sus características.
22 Paquetes integrados: Lotus 1-2-3 y Simphony
Estudio de dos de los paquetes integrados (Hoja de cálculo + base de datos + ...) más conocidos.
23 dBASE II y dBASE III
Cómo aprovechar las dos versiones más recientes de esta importante base de datos.
24 Los ordenadores uno a uno
Un amplio y completo estudio comparativo.
25 Cálculo numérico en BASIC
Una aplicación especializada a su disposición.
26 Multiplan
Cómo hacer uso de este moderno paquete de aplicaciones.
27 FORTRAN y COBOL
Dos lenguajes muy especializados y distintos.
28 FORTH: anatomía de un lenguaje inteligente
Principales características de un lenguaje moderno, flexible y de amplio uso, en la robótica.
29 Cómo realizar nuestro propio banco de datos
Conocimientos necesarios para poder fabricar un banco de datos a nuestro gusto y medida.
30 Los paquetes integrados uno a uno
Todos los que usted puede encontrar en el mercado.

NOTA: Ingelek, S. A. se reserva el derecho de modificar, sin previo aviso, el orden, título o contenido de cualquier volumen de la colección.

NOTAS



EL problema de la construcción racional y bien organizada de un programa surgió bastante pronto en la historia de la informática. La prisa por desarrollar una aplicación por un lado y, por otro, la anarquía de los programadores han provocado a menudo auténticos desastres, plasmados normalmente en la escasa legibilidad y la costosa manutención del software (por no hablar de los errores bug, más o menos difíciles de localizar).

La programación estructurada nació precisamente con el fin de poner orden en el caos, tratando de imponer un estilo y un planteamiento claros y metódicos, basados en reglas simples y precisas

Los lenguajes estructurados (de los que, en el ámbito de los ordenadores personales, el más conocido es el Pascal) son algo mucho más profundo y, a la vez, más potente que unas simples estructuras de control. Fundamentalmente, se trata de la subdivisión en bloques (en niveles jerárquicos) y de la definición racional de los tipos de datos. Se descubren así otros conceptos complejos, pero muy útiles, como la recursividad.

Basándonos en el Pascal, podremos entrar en la programación estructurada y conseguir, por lo menos, una estructuración mental que nos permita hacer un programa como es debido.

375 pts.